Chapter 3 : Memory Management-2

Choi Yong-jae
bestjae@naver.com

Initailization of Memory Management

- Data Structure Setup
- Architecture-Specific Setup
- Memory Management During the Boot Process

Data Structure Setup

- Prerequisites
  - the kernel defines a single instance of pg_data_t in mm/page_alloc.c

<mmzone.h>

```
#define NODE_DATA(nid)          (&contig_page_data)
```

```
typedef struct pglist_data {
    struct zone node_zones[MAX_NR_ZONES];
    struct zonelist node_zonelists[MAX_ZONELISTS];
    int nr_zones;
#ifdef CONFIG_FLAT_NODE_MEM_MAP /* means !SPARSEMEM */
    struct page *node_mem_map;
#ifdef CONFIG_PAGE_EXTENSION
    struct page_ext *node_page_ext;
#endif
#endif
#ifndef CONFIG_NO_BOOTMEM
    struct bootmem_data *bdata;
#endif
#ifdef CONFIG_MEMORY_HOTPLUG
```

Data Structure Setup

- System Start

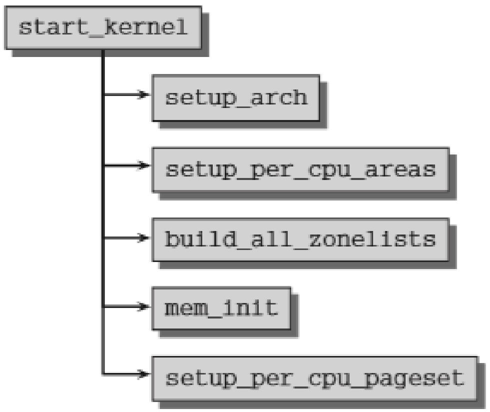

Figure 3-8: Kernel initialization in the view of memory management.

```
boot_cpu_init();
page_address_init();
pr_notice("%s", linux_banner);
setup_arch(&command_line);
mm_init_cpumask(&init_mm);
setup_command_line(command_line);
setup_nr_cpu_ids();
setup_per_cpu_areas();
boot_cpu_state_init();
smp_prepare_boot_cpu(); /* arch-specific boot-cpu hooks */

build_all_zonelists(NULL, NULL);
page_alloc_init();
```

  - *setup_arch* is an architecture-specific set-up function responsible for, among other things, initialization of the bootallocator.
  - *setup_per_cpu_areas* initializes per-CPU variables defined statically in the source cod

Data Structure Setup

- Node and Zone Initialization
  - *build_all_zonelists* builds the data structures required to manage nodes and their zones
  - it can be implemented by the macros and abstraction mechanisms introduced above regardless of whether it runs on a NUMA or UMA system
  - delegates all work to __build_all_zonelists, which, in turn, invokes build_zonelists for each NUMA node in the system.

```
static int __build_all_zonelists(void *data)
{
    int nid;
    int cpu;
    pg_data_t *self = data;

#ifdef CONFIG_NUMA
    memset(node_load, 0, sizeof(node_load));
#endif

    if (self && !node_online(self->node_id)) {
        build_zonelists(self);
    }

    for_each_online_node(nid) {
        pg_data_t *pgdat = NODE_DATA(nid);

        build_zonelists(pgdat);
    }
```

```
static void build_zonelists(pg_data_t *pgdat)
{
    int i, node, load;
    nodemask_t used_mask;
```

```
if (order == ZONELIST_ORDER_ZONE) {
    /* calculate node order -- i.e., DMA last!
    build_zonelists_in_zone_order(pgdat, i);
}
```

Data Structure Setup

- Node and Zone Initialization
  - *build_zonelists* 노드 메모리 구성 및 새롭게 생성 된 데이터 구조를 유지하는 정보를 포함하는 데이터 인스턴스를 저장
  - 현재 처리중인 노드의 영역과 시스템의 다른 노드 사이의 순위 결정
  - 우선 순위에 따라 메모리 할당

  - 커널이 높은 메모리를 할당
    - 현재 노드의 highmem 영역에서 적절한 크기의 세그먼트 찾음
    - 실패하면 노드의 일반 메모리 영역 확인
    - 또 실패시 노드의 DMA 영역에서 할당을 시도. 세 개의 로컬 영역 중 하나에서도 여유 공간을 찾을 수없는 경우 다른 노드를 찾습니다. 이 경우 대체 노드는 비 로컬 메모리에 액세스 한 결과로 인한 성능 손실을 최소화하기 위해 가능한 한 주 노드에 가깝게해야합니다.
  - The kernel defines a memory hierarchy and first tries to allocate “cheap“ memory

Data Structure Setup

- Node and Zone Initialization
  - The kernel uses an array of zonelist elements in pg_data_t to represent the described hierarchy as a data structure.

<mmzone.h>

```
typedef struct pglist_data {
  ...
        struct zonelist node_zonelists[MAX_ZONELISTS];
...
} pg_data_t;

#define MAX_ZONES_PER_ZONELIST (MAX_NUMNODES * MAX_NR_ZONES)
struct zonelist {
...
        struct zone *zones[MAX_ZONES_PER_ZONELIST + 1];    // NULL delimited
};
```

Data Structure Setup

- Node and Zone Initialization
  - node_zonelists array makes a separate entry available for every possible zone type.
  - fallback hierarchy is delegated to *build_zonelists*

```
static int build_zonelists_node(pg_data_t *pgdat, struct zonelist *zonelist,
                int nr_zones)
{
    struct zone *zone;
    enum zone_type zone_type = MAX_NR_ZONES;

    do {
        zone_type--;
        zone = pgdat->node_zones + zone_type;
        if (managed_zone(zone)) {
            zoneref_set_zone(zone,
                &zonelist->_zonerefs[nr_zones++]);
            check_highest_zone(zone_type);
        }
    } while (zone_type);

    return nr_zones;
}
```

Data Structure Setup

- Node and Zone Initialization
  - The kernel must then establish the order in which the zones of the other nodes in the system are used as fallback targets.

mm/page_alloc.c

```
static void __init build_zonelists(pg_data_t *pgdat)
{
  ...
                for (node = local_node + 1; node < MAX_NUMNODES; node++) {
                        j = build_zonelists_node(NODE_DATA(node), zonelist, j, i);
                }
                for (node = 0; node < local_node; node++) {
                        j = build_zonelists_node(NODE_DATA(node), zonelist, j, i);
                }

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

Data Structure Setup

- Node and Zone Initialization
  - The kernel must then establish the order in which the zones of the other nodes in the system are used as fallback targets.

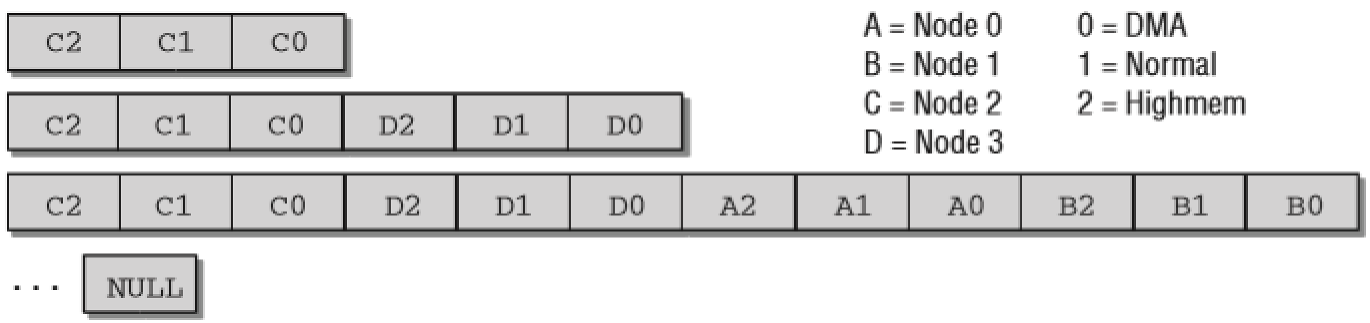

Figure 3-9: Successive filling of the fallback list.

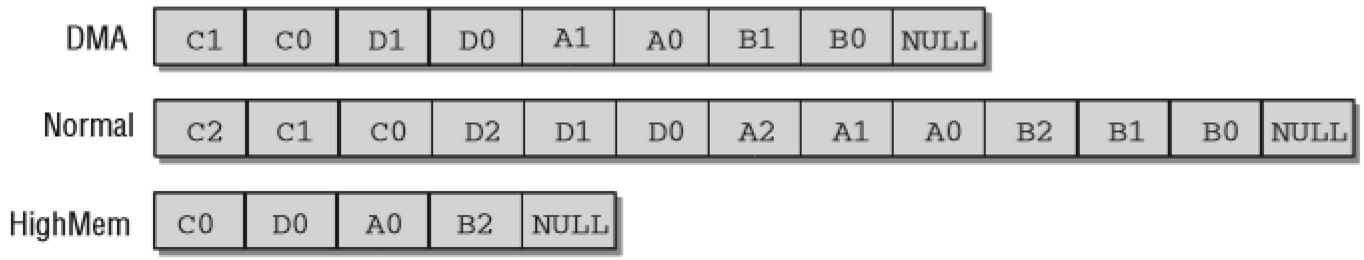

Figure 3-10: Finished fallback lists.

Architecture-Specific Setup

- Arrangement of the kernel in memory
- Initialization Steps
- Initialization of Paging
- Registering Active Memory Regions
- Address Space Setup on AMD64

Architecture-Specific Setup

- Arrangement of the kernel in memory
  - boot loader has copied the kernel into memory and the assembler part of the initialization routines
  - It is also possible to configure the initial position of the kernel binary in physical RAM if the crash dump mechanism is enabled
  - IA-32 kernels use 0x100000 as the start address

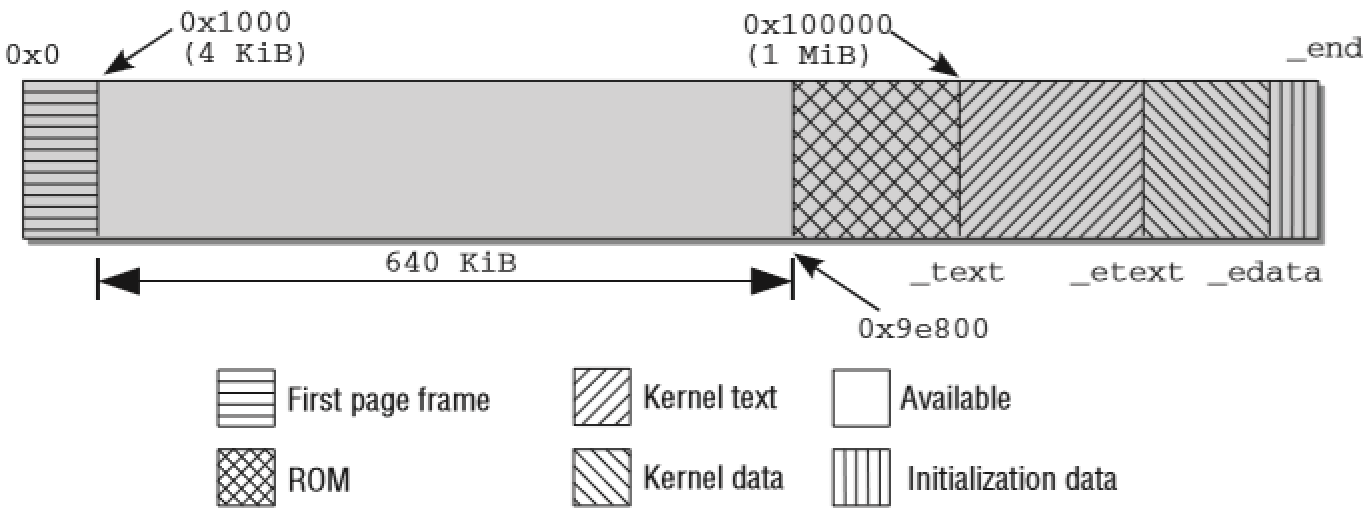

Figure 3-11: Arrangement of the Linux kernel in RAM memory.

Architecture-Specific Setup

- Arrangement of the kernel in memory
  - System.map is generated and stored in the source base directory.
  - the addresses of all other (global) variables, procedures, and functions defined in the kernel

```
wolfgang@meitner> cat System.map
...
c0100000 A _text
...
c0381ecd A _etext
...
c04704e0 A _edata
...
c04c3f44 A _end
```

  - /proc/iomem also provides information on the sections into which RAM memory is divided.

```
wolfgang@meitner> cat /proc/iomem
00000000-0009e7ff : System RAM
0009e800-0009ffff : reserved
000a0000-000bffff : Video RAM area
000c0000-000c7fff : Video ROM
000f0000-000fffff : System ROM
00100000-17ceffff : System RAM
  00100000-00381ecc : Kernel code
  00381ecd-004704df : Kernel data
```

Architecture-Specific Setup

- Initialization Steps
  - Recall that setup_arch is invoked from within start_kernel
  - machine_specific_memory_setup은 먼저 시스템이 차지하는 메모리 영역과 여유 메모리 영역이있는 목록을 생성하기 위해 호출됩니다
  - When the system is booted, the regions found are displayed by the kernel function print_memory_map.

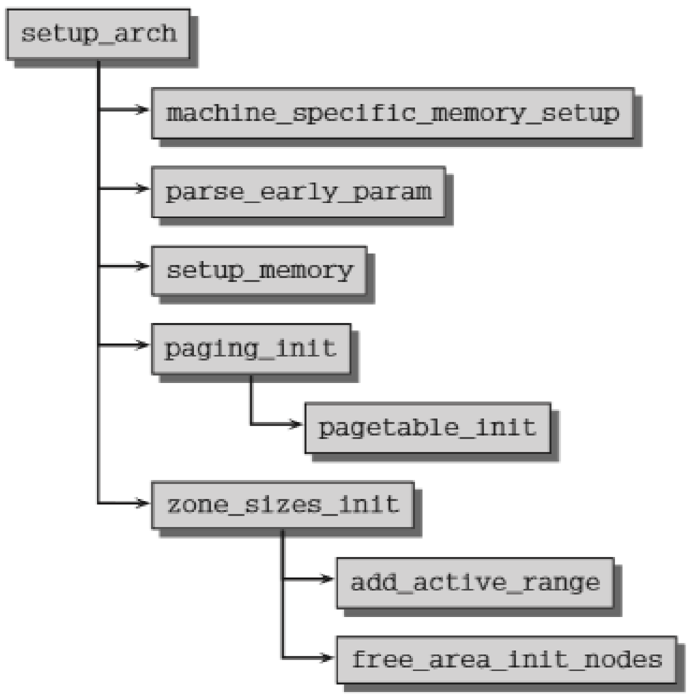

Figure 3-12: Code flow diagram for memory initialization on IA-32 systems.

Architecture-Specific Setup

- Initialization Steps
  - *Setup_memory* – Two versions
    - contiguous memory (in arch/x86/kernel/setup_32.c)
    - discontiguous memory (in arch/x86/mm/discontig_32.c)
    - The number of physical pages available (pernode) is determined
    - The bootmem allocator is initialized (Section 3.4.3 describes the implementation of the allocator in detail)
    - Various memory areas are then reserved, for instance, for the initial RAM disk needed when running the first userspace processe
  - *Paging_init*
    - initializes the kernel page tables and enables paging
  - *Pagetable_init*
    - ensures that the direct mapping of physical memory into the kernel address space is initialized
  - *zone_sizes_init*
    - initializes the pgdat_t instances of all nodes of the system

Architecture-Specific Setup

- Initialization of Paging
  - kernel typically divides the total available virtual address space of 4 GiB in a ratio of 3 : 1
    - the kernel must be embedded in a reliable environment
    - <u>The physical pages are mapped to the start of the kernel address space so that the kernel can access them directly without the need for complicated page table operations.</u>

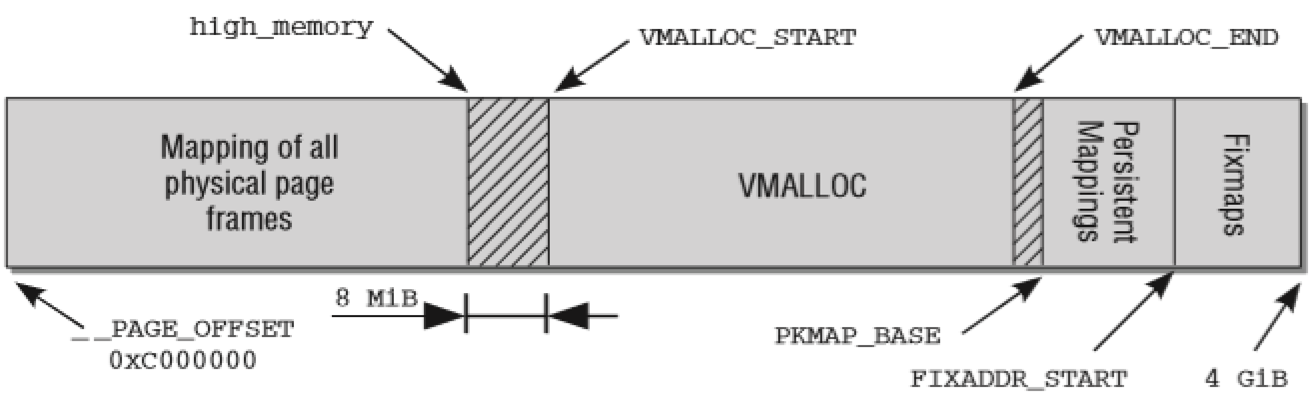

Figure 3-15: Division of the kernel address space on IA-32 systems.

Architecture-Specific Setup

- Initialization of Paging
  - 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
  - kernel port must provide two macros for each architecture to translate between physical and virtual addresses
    - __pa(vaddr) returns the physical address associated with the virtual
    - __va(paddr) yields the virtual address corresponding to the physical address paddr.
  - kernel use the last 128 MiB of its address space
    - Virtually contiguous memory areas that are not contiguous in physical memory can be reserved in the vmalloc area.
    - Persistent mappings are used to map non-persistent pages from the highmem area into the kernel
    - Fixmaps are virtual address space entries associated with a fixed but freely selectable page in physical address space

Architecture-Specific Setup

- Initialization of Paging
  - There is a gap with a minimum size of VMALLOC_OFFSET between the direct mapping of all RAM pages and the area for non-contiguous allocations.
    - safeguard against any kernel faults
  - VMALLOC_START and VMALLOC_END define the start and end of the vmalloc area used for physically noncontiguous kernel mappings

include/asm-x86/pgtable_32.h

```
#define VMALLOC_START    (((unsigned long) high_memory + \
                         2*VMALLOC_OFFSET-1) & ~(VMALLOC_OFFSET-1))
#ifdef CONFIG_HIGHMEM
# define VMALLOC_END     (PKMAP_BASE-2*PAGE_SIZE)
#else
# define VMALLOC_END     (FIXADDR_START-2*PAGE_SIZE)
#endif
```

  - start address of the vmalloc area depends on how much virtual address space memory is used for the direct mapping of RAM

Table 3-5: VMALLOC_OFFSET **Values for Different RAM Sizes**

Memory (MiB)	Offset (MiB)
128	8
129	15

Architecture-Specific Setup

- Initialization of Paging
  - Last memory section is occupied by fixed mappings
    - Point to a random location in RAMmemory
  - The advantage of fixmap addresses is that at compilation time
    - the address acts like a constant whose physical address is assigned when the kernel is booted
  - Alternative Division
    - it may be better to split the address space symmetrically, 2 GiB for user address space and 2 GiB for kernel address space
      - __PAGE_OFFSET must then be set to 0x80000000 instead of the typical default of 0xC0000000
  - Splitting the Virtual Address Space
    - *paging_init* is invoked on IA-32 systems during the boot process to split the virtual address space as described above.

Architecture-Specific Setup

- Initialization of Paging
  - Pagetable_init first initializes the page tables of the system using swapper_pg_dir as a basic
  - Initialization of the Hot-n-ColdCache
    - *zone_pcp_init* is responsible for initializing the cache
    - Batch size has been determined with *zone_bachsize*

mm/page_alloc.c

```
static __devinit void zone_pcp_init(struct zone *zone)
{
        int cpu;
        unsigned long batch = zone_batchsize(zone);

        for (cpu = 0; cpu < NR_CPUS; cpu++) {
                setup_pageset(zone_pcp(zone,cpu), batch);
        }
        if (zone->present_pages)
                printk(KERN_DEBUG " %s zone: %lu pages, LIFO batch:%lu\n",
                        zone->name, zone->present_pages, batch);
}
```

Architecture-Specific Setup

- Registering Active Memory Regions
  - An active memory region is simply a memory region that does not contain any holes
  - individual architectures can still decide to set up all data structures on their own without relying on the generic framework provided by the kernel.
  - generic framework must set the configuration option ARCH_POPULATES_NODE_MAP

arch/x86/kernel/setup_32.c

```
void __init zone_sizes_init(void)
{
        unsigned long max_zone_pfns[MAX_NR_ZONES];
        memset(max_zone_pfns, 0, sizeof(max_zone_pfns));
        max_zone_pfns[ZONE_DMA] =
                virt_to_phys((char *)MAX_DMA_ADDRESS) >> PAGE_SHII
        max_zone_pfns[ZONE_NORMAL] = max_low_pfn;
#ifdef CONFIG_HIGHMEM
        max_zone_pfns[ZONE_HIGHMEM] = highend_pfn;
        add_active_range(0, 0, highend_pfn);
#else
        add_active_range(0, 0, max_low_pfn);
 #endif

        free_area_init_nodes(max_zone_pfns);
}
```

arch/x86/kernel/e820_64.c

```
e820_register_active_regions(int nid, unsigned long start_pfn,
                                                                unsigned
{
        unsigned long ei_startpfn;
        unsigned long ei_endpfn;
        int i;

        for (i = 0; i < e820.nr_map; i++)
        if (e820_find_active_region(&e820.map[i],
                                    start_pfn, end_pfn,
                                    &ei_startpfn, &ei_endpfn))
                add_active_range(nid, ei_startpfn, ei_endpfn);
}
```

Architecture-Specific Setup

- Address Space Setup on AMD64
  - The address space spanned by 64 bits is so large that there are currently simply no applications that would require this
  - Current implementations therefore implement a smaller physical address space that is only 48 bits wide
  - They divide the total address space into three parts: a lower half, a higher half, and a forbidden region in between.

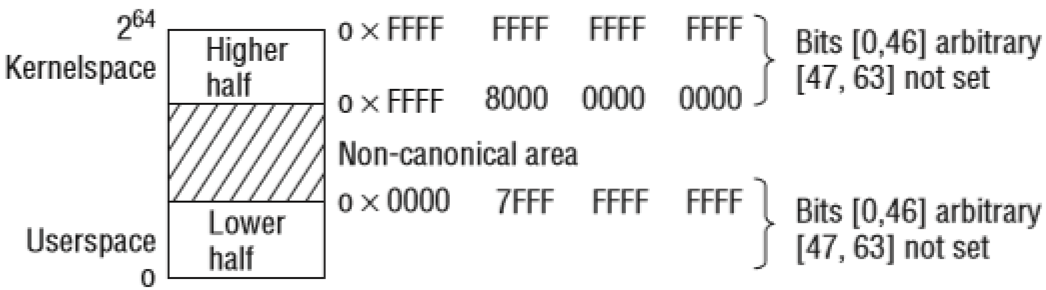

Figure 3-18: Possible virtual versus implemented physical address space on AMD64 machines.

Q & A

Q & A