Part 04. 리눅스 커널 입문

Chapter 03. 메모리

진행 순서

Chapter 03_01	물리 메모리 관리
Chapter 03_02	커널 메모리 할당
Chapter 03_03	버디 (Buddy) 와 슬랩 (Slab)
Chapter 03_04	슬랩 캐시

Chapter 03_01 물리 메모리 관리

키워드

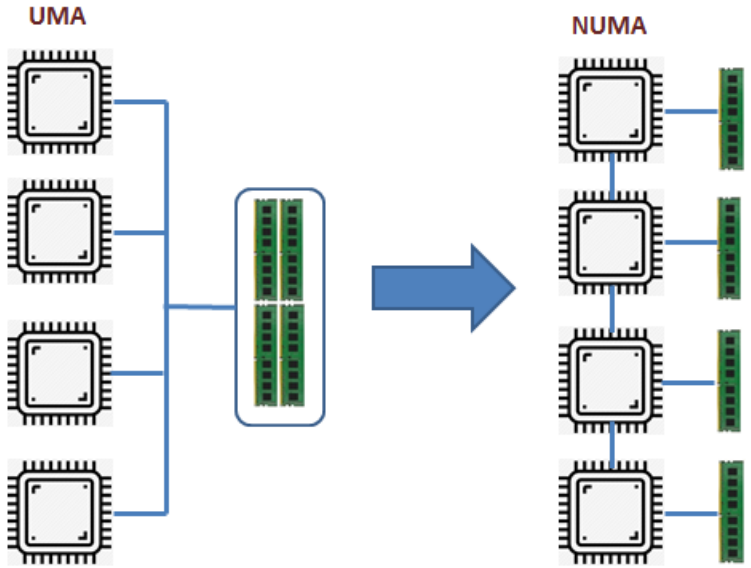
SMP (Symmetric Multiprocessing): 다수의 CPU가 동일한 메모리를 공유할 수 있는 다중처리 구조

UMA (Uniform Memory Access) : 균일 기억장치 접근

- 모든 프로세서들이 상호간에 연결되어 하나의 메모리를 공유하는 기술

NUMA (Non-Uniform Memory Access): 불균일 기억장치 접근

- 효율성, 병목 현상 해결
- 메모리에 접근하는 시간이 프로세서와 메모리의 상대적 위치에 따라서 달라짐
- 클러스터링 개념



Chapter 03_01 물리 메모리 관리

노드 (Node): 다수의 메모리 집합, NUMA 구조에선 다수의 메모리 노드가 존재
- (/include/linux/mmzone.h)
- pgdat_list 이름의 배열을 통해 접근
- 하나의 노드는 pg_data_t 구조체를 통해서 표현

```
struct pglist_data *pgdat_list[];

typedef struct pglist_data {
        struct zone node_zones[MAX_NR_ZONES];
        struct zonelist node_zonelists[MAX_ZONELISTS];
        int nr_zones;
        struct page *node_mem_map;
        struct bootmem_data *bdata;
        unsigned long node_start_pfn;
        unsigned long node_present_pages; /* total number of physical pages */
        unsigned long node_spanned_pages; /* total size of physical page range, including holes */
        int node_id;
        wait_queue_head_t kswapd_wait;
        wait_queue_head_t pfmemalloc_wait;
        struct task_struct *kswapd;          /* Protected by mem_hotplug_begin/end() */
        …
        unsigned long                        flags;
        …
} pg_data_t;
```

Chapter 03_01 물리 메모리 관리

zone: 메모리 내의 범위를 나타내는 노드 내의 블록
- ZONE_DMA/ZONE_DMA32 (0 ~ 16M) : ISA 버스 기반 디바이스에서 사용
- ZONE_NORMAL (16 ~ 896M) : 커널의 가상주소공간과 1:1 연결
- ZONE_HIGHMEM (896 ~ End) : 필요할 때 동적으로 연결
- /proc/zoneinfo 참조
- (/include/linux/mmzone.h)

```
struct zone {
        unsigned long watermark[NR_WMARK];
        unsigned long nr_reserved_highatomic;
        long lowmem_reserve[MAX_NR_ZONES];
        int node;
        struct pglist_data          *zone_pgdat;
        struct per_cpu_pageset __percpu *pageset;
        unsigned long                   zone_start_pfn;

        unsigned long                   managed_pages;
        unsigned long                   spanned_pages;
        unsigned long                   present_pages;

        const char                  *name;
        …
} ____cacheline_internodealigned_in_smp;
```

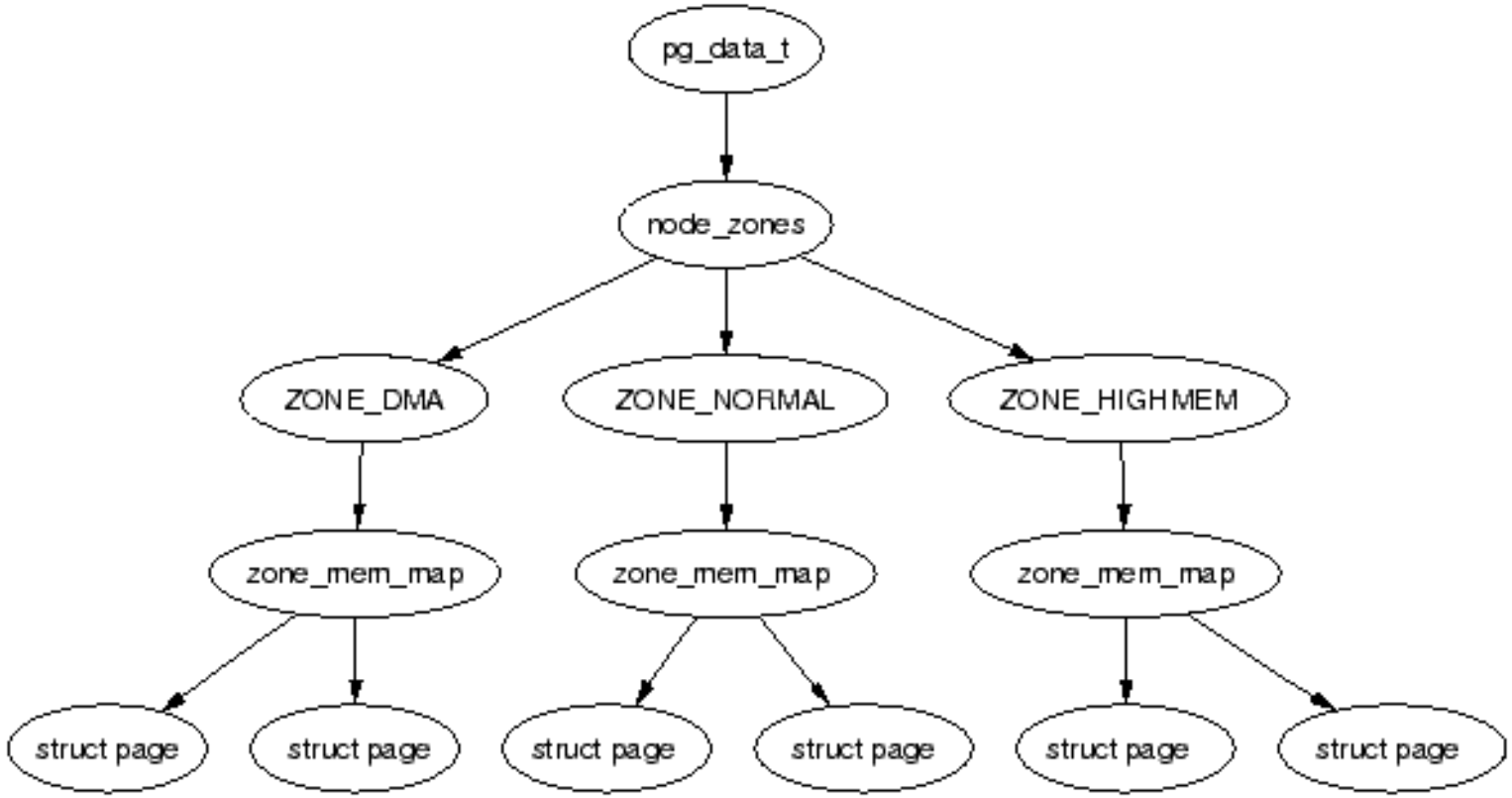
```
[parallels@localhost ~]$ cat /proc/zoneinfo
Node 0, zone      DMA
  pages free     2471
        min      95
        low      118
        high     142
        scanned  0
        spanned  4095
        present  3998
        managed  3977
    nr_free_pages 2471
    nr_alloc_batch 22
    nr_inactive_anon 28
    nr_active_anon 580
    nr_inactive_file 397
    nr_active_file 148
    nr_unevictable 0
    nr_mlock     0
    nr_anon_pages 575
    nr_mapped    179
    nr_file_pages 578
    nr_dirty     0
    nr_writeback 0
    nr_slab_reclaimable 70
    nr_slab_unreclaimable 103
    nr_page_table_pages 65
    nr_kernel_stack 1
    nr_unstable  0
```

Chapter 03_01 물리 메모리 관리

page: 물리 메모리의 최소 단위를 페이지 프레임(page frame)이라 부른다.
- 각각의 zone은 자신에 속해 있는 페이지 프레임을 관리
- (/include/linux/mm_types.h)
- page 구조체
- mem_map 이라는 전역 배열을 통해 접근

```
struct page {
   unsigned long flags;       /* Atomic flags, some possibly updated asynchronously */
   union {
      struct address_space *mapping;
      void *s_mem;          /* slab first object */
      atomic_t compound_mapcount;    /* first tail page */
   };
   union {
      pgoff_t index;       /* Our offset within mapping. */
      void *freelist;       /* sl[aou]b first free object */
   };
   union {
      unsigned counters;
      struct {
         union {
            atomic_t _mapcount;
…
}
```

Chapter 03_01 물리 메모리 관리



Chapter 03_02 커널 메모리 할당

```
#include <linux/slab.h>

void *kmalloc(size_t size, gfp_t flags);
void kfree(const void *ptr);
```

최소한 size 바이트 크기를 가진 메모리 영역의 포인터를 반환합니다.
할당된 메모리 영역은 물리적으로 연속된 영역입니다.
사용가능한 메모리가 부족한 경우 NULL을 반환합니다.

flags	
GFP_KERNEL	중단 가능한 일반적인 할당 (프로세스 컨택스트)
GFP_ATOMIC	휴면 상태로 전환하면 안되는 할당 (인터럽트 핸들러, 후반부 처리 작업, 스핀락 구간)
…	

ex)

```
struct box *p;
p = kmalloc(sizeof(struct box), GFP_KERNEL);
if (!p)
   /* error */
kfree(p);
```

Chapter 03_02 커널 메모리 할당

```
#include <linux/slab.h>

void *vmalloc(unsigned long size);
void vfree(const void *addr);
```

물리적으로 연속될 필요없이 가상적으로 연속된 메모리 영역을 할당
반환하는 페이지는 가상 주소 공간에서 연속된 공간이지만,
이 공간이 실제 물리적인 메모리에서도 연속적이라는 보장은 없습니다.
일반적으로 큰 사이즈 메모리 할당 시 사용, kmalloc 대비 속도가 느림
휴면 상태로 전환이 가능하며, 인터럽트 컨택스트와 같이 작업 도중 휴면이 불가능한 상황에서는 사용할 수 없습니다.

ex)

```
char *buf;
buf = vmalloc(16 * PAGE_SIZE);
if (!buf)
   /* error */
vfree(buf);
```

Chapter 03_03 버디 (Buddy) 와 슬랩 (Slab)

리눅스 커널은 메모리 할당 시 단편화 문제를 해결하기 위해 버디와 슬랩 시스템을 도입

버디 (Buddy) : 외부 단편화 해결
- 물리 메모리의 최소 단위인 페이지 4KB 사이즈
- 4K의 배수 단위로 미리 묶어 놓음 (4KB, 8KB, 16KB, …)
- ex. 페이지 사이즈보다 큰 10KB 메모리 요청 시 미리 묶어 놓은 16KB 버디 할당
- 2.6.19부터는 Lazy Buddy 개념 도입
- 큰 페이지 쪼개서 할당/합쳐서 관리 하는 반복을 최소화, 합치는 작업을 뒤로 미룸

슬랩 (Slab): 내부 단편화 해결
- 페이지 사이즈 보다 작은 메모리 요청 시 사용
- 페이지 프레임을 작은 사이즈로 분할하여 요청 시 할당
- 작은 사이즈의 캐시(cache) 집합을 통해 메모리를 관리
- 32Byte, 64Byte, 128Byte, …, 4MB
- kmem_cache_alloc(), kmem_cache_free()
- 물리적으로 연속적인 공간

Chapter 03_04 슬랩 캐시(Slab Cache)

자료구조를 다 사용하고 난 뒤 메모리를 해제하는 대신 해제 리스트에 추가해 놓고
새로운 자료구조를 할당하는 경우 새로 메모리를 할당하고 그 메모리에 자료구조를 준비하는 대신
해제 리스트에 들어있는 자료구조를 바로 사용할 수 있다.

- 사용이 빈번한 객체 캐시(Object Cache)
- 리눅스 커널은 범용 자료구조 캐시 계층인 슬랩 계층을 제공
- 슬랩 계층은 각 객체를 유형별로 객체를 저장하는 캐시(Cache)에 분류한다.

Chapter 03_04 슬랩 캐시(Slab Cache)

캐시 할당과 제거

```
struct kmem_cache *kmem_cache_create(const char *name, size_t size, size_t align, unsigned long flags, void (*ctor)(void *));
```

kmem_cache_create()를 통해 새로운 캐시를 만들 수 있습니다.
name – 캐시의 이름을 저장
size – 캐시에 들어갈 항목의 크기
align – 슬랩 내부의 첫 번째 객체의 오프셋, 0 (표준정렬)
flags – 캐시의 동작을 제어하기 위한 설정, 특별한 동작이 필요 없는 경우 0
ctor – 캐시 생성자, 캐시에 새로운 페이지가 추가될 때마다 호출된다, 필요 없는 경우 NULL

```
int kmem_cache_destroy(struct kmem_cache *cachep);
```

kmem_cache_destroy()는 지정한 캐시를 제거합니다.
- 캐시의 모든 슬랩이 비어 있어야 합니다.
- 호출하는 동안 캐시에 접근하는 경우가 있어서는 안됩니다.

Chapter 03_04 슬랩 캐시(Slab Cache)

캐시 내부 객체 할당과 해제

```
void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags);
```

캐시를 만든 후 kmem_cache_alloc()을 통해서 캐시 내부의 객체를 얻을 수 있습니다.
cachep가 지정한 캐시에 들어있는 객체의 포인터를 반환합니다.
flags는 GFP_KERNEL 또는 GFP_ATOMIC

```
void kmem_cache_free(struct kmem_cache *cachep, void *objp);
```

kmem_cache_free()를 통해 사용한 객체를 해제하고, 슬랩에 반환합니다.

Chapter 03_04 슬랩 캐시(Slab Cache)

kmem_cache 사용 예 (kernel/fork.c)

```
struct kmem_cache *task_struct_cachep; /* 전역 변수 */

task_struct_cachep = kmem_cache_create(“task_struct”, sizeof(struct task_struct), ARCH_MIN_TASKALIGN,
   SLAB_PANIC | SLAB_NOTRACK, NULL);    /* struct task_struct 객체를 저장하는 task_struct라는 이름의 캐시가 만들어 짐 */

/* 프로세스가 fork()를 호출할 때마다 새로운 프로세스 서술자가 만들어진다. */
struct task_struct *tsk;

tsk = kmem_cache_alloc(task_struct_cachep, GFP_KERNEL);
if (!tsk)
   return NULL;

/* 태스크가 종료된 다음 자식 프로세스가 없으면 프로세스 서술자가 해제, 슬랩 캐시에 반환 */
kmem_cache_free(task_struct_cachep, tsk);

/* 프로세스 서술자는 커널의 핵심 부분이고 항상 필요하므로 캐시는 해제되지 않음 */
```