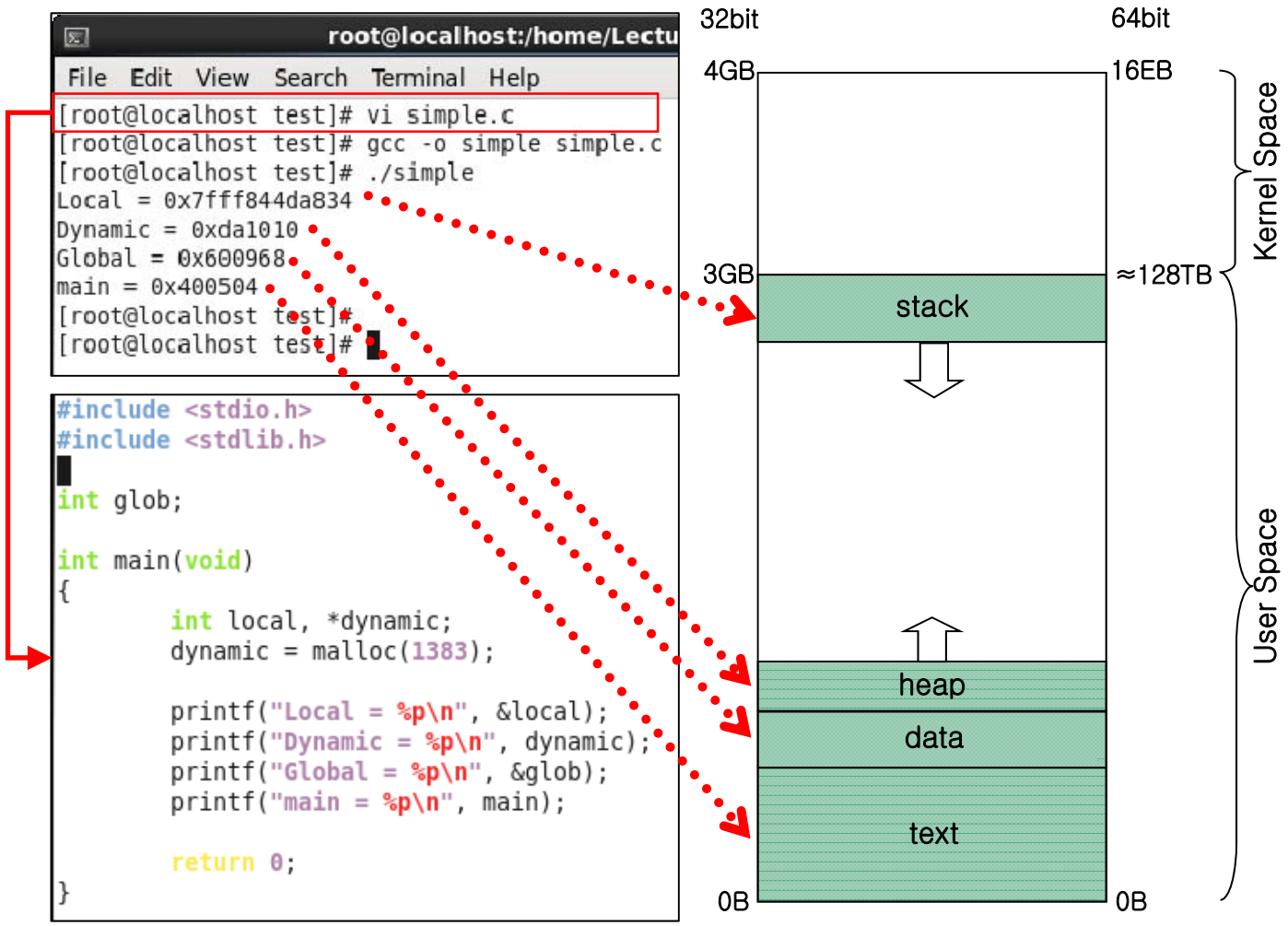
Memory Management

April, 2016
Seungjae Baek

Dept. of software
Dankook University
http://embedded.dankook.ac.kr/~baeksj

Program, Process & Virtual Address



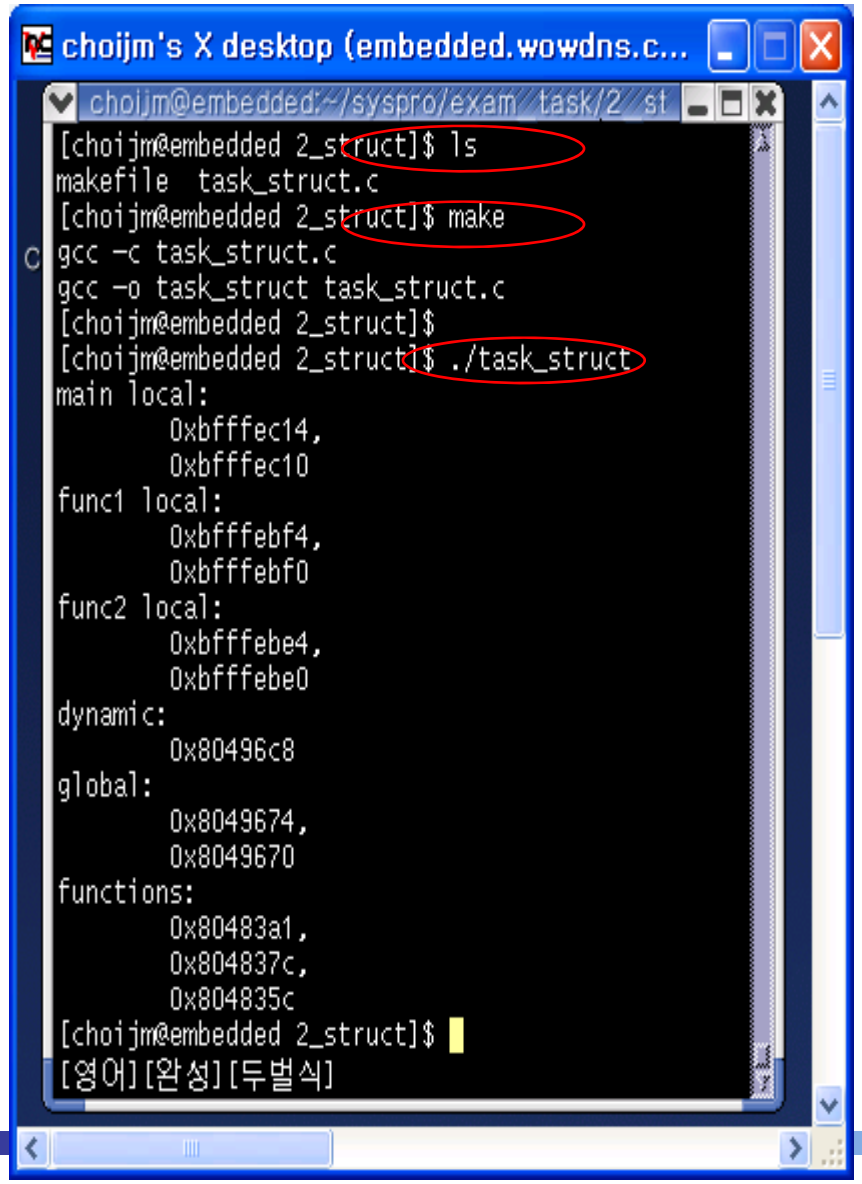
가상 주소 확인

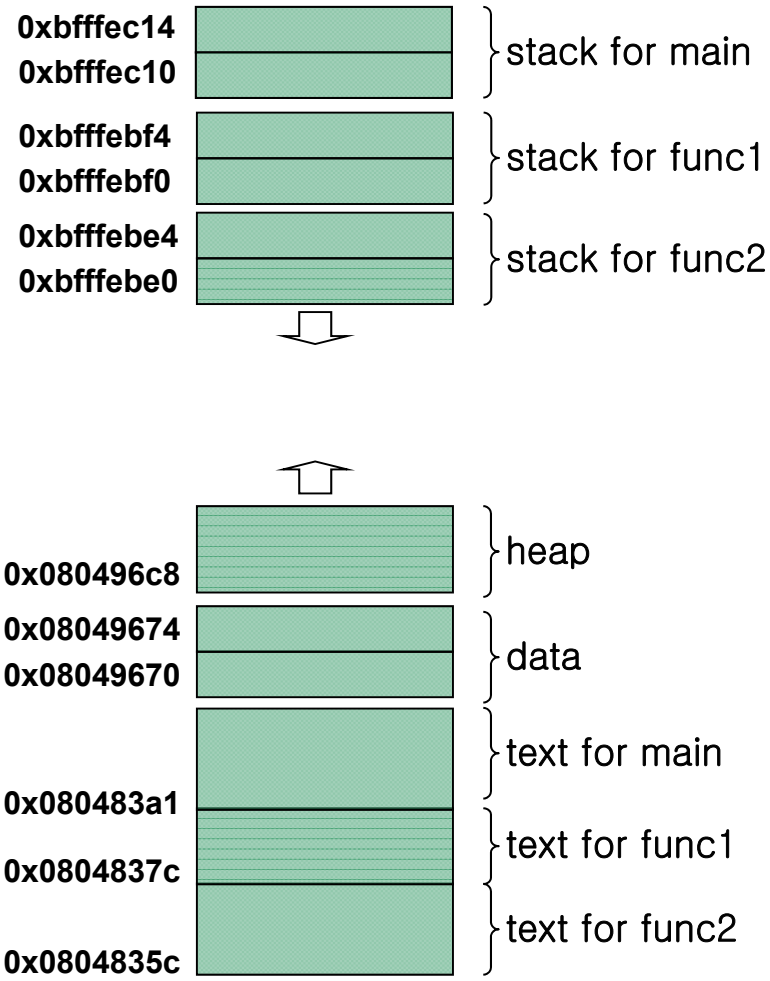
- C 프로그램에서 본 태스크 이미지: 주소 출력

```
#include <stdlib.h>

int glob1, glob2;

int func2() {
    int f2_local1, f2_local2;

    printf("func2 local: ₩n₩t%p, ₩n₩t%p₩n", &f2_local1, &f2_local2);
}

int func1() {
    int f1_local1, f1_local2;

    printf("func1 local: ₩n₩t%p, ₩n₩t%p₩n", &f1_local1, &f1_local2);
    func2();
}

main()
{
    int m_local1, m_local2; int *dynamic_addr;

    printf("main local: ₩n₩t%p, ₩n₩t%p₩n", &m_local1, &m_local2);
    func1();

    dynamic_addr = malloc(16);
    printf("dynamic: ₩n₩t%p₩n", dynamic_addr);
    printf("global: ₩n₩t%p, ₩n₩t%p₩n", &glob2, &glob1);
    printf("functions: ₩n₩t%p, ₩n₩t%p, ₩n₩t%p₩n", main, func1, func2);
}
```

가상 주소 확인 결과

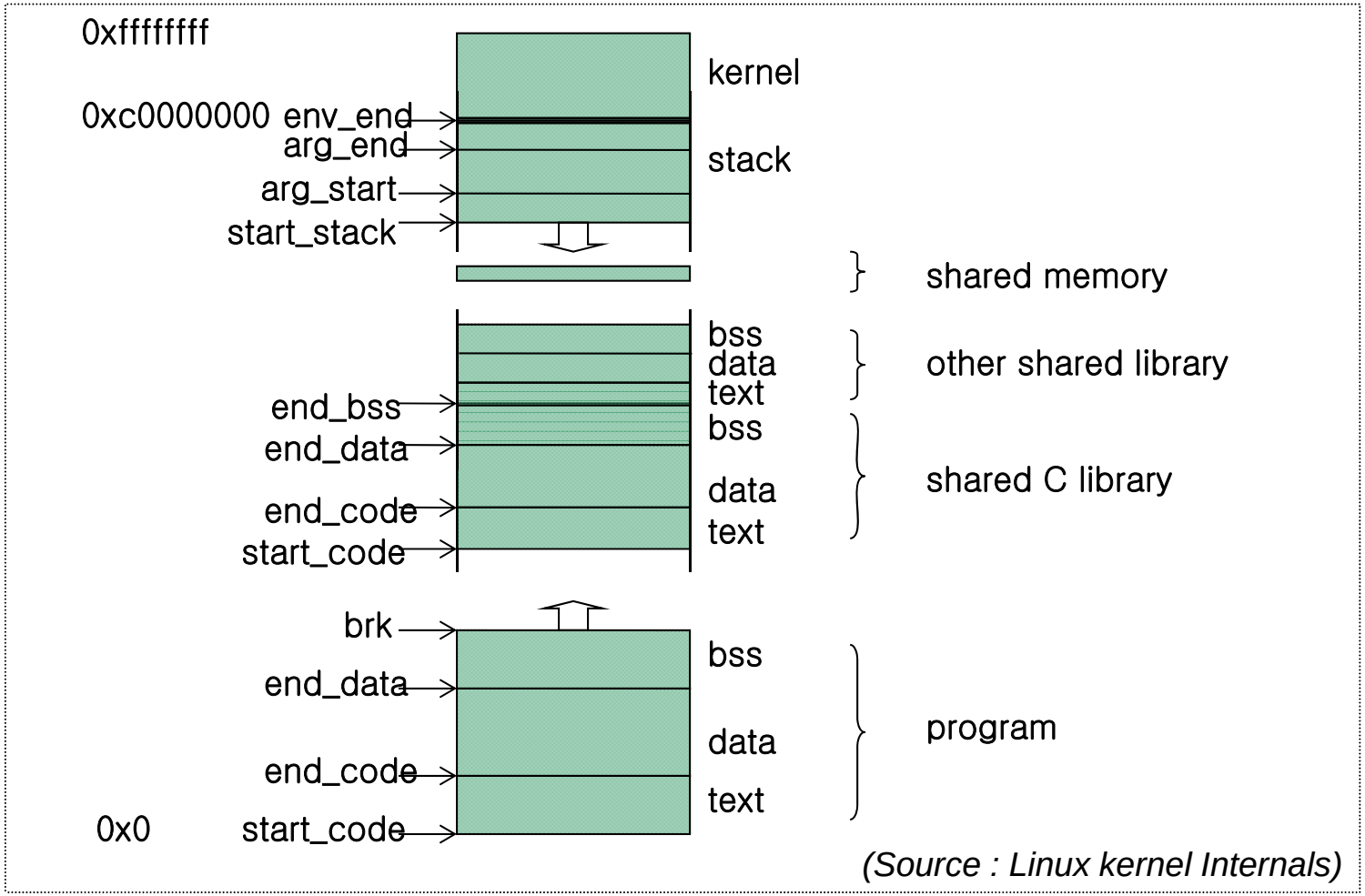
- C 프로그램에서 본 태스크 이미지: 주소 출력 결과





가상 메모리 관리 구조

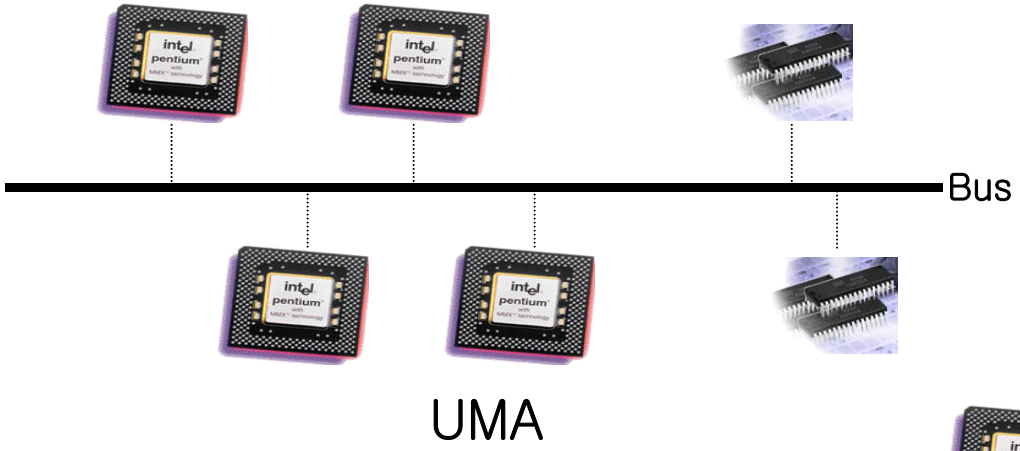
- Linux 커널에서 가상 메모리 구조

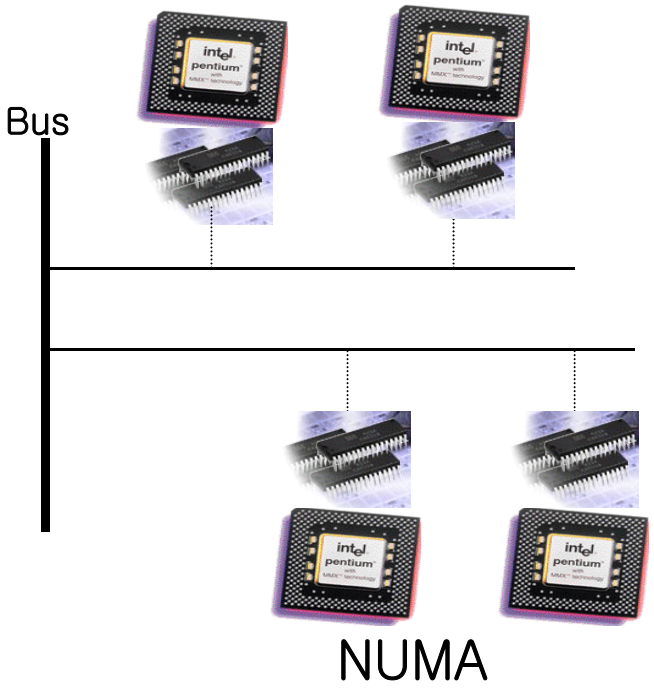


메모리 관리 기법

- 물리 메모리 관리
  - ✓ 할당 / 해제 기법
- 가상 메모리 관리
  - ✓ 할당 / 해제 기법
- 물리 메모리와 가상 메모리
  - ✓ 연결 혹은 변환 기법

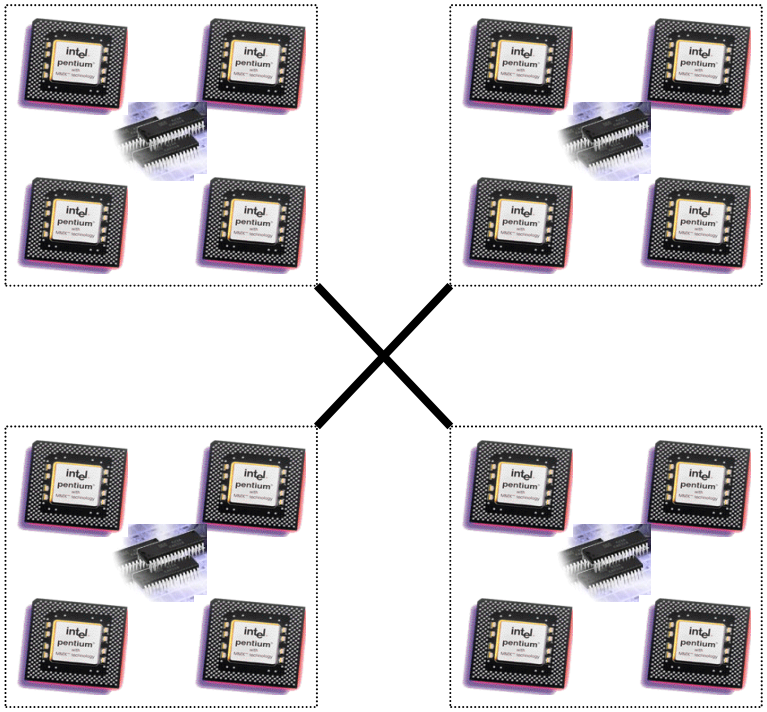
Memory Model(1/2)



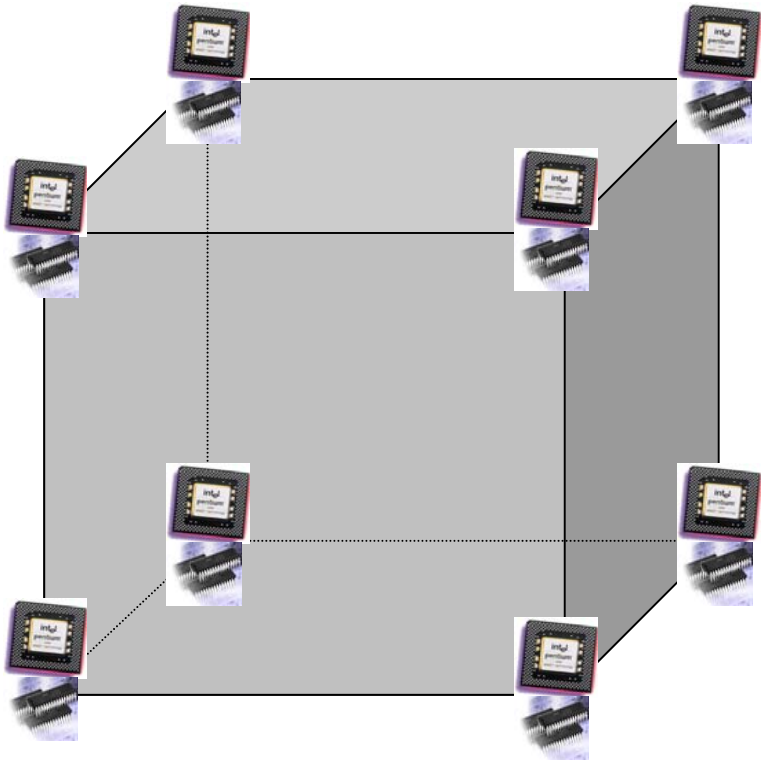


(Source : Unix Internals)

Memory Model(2/2)

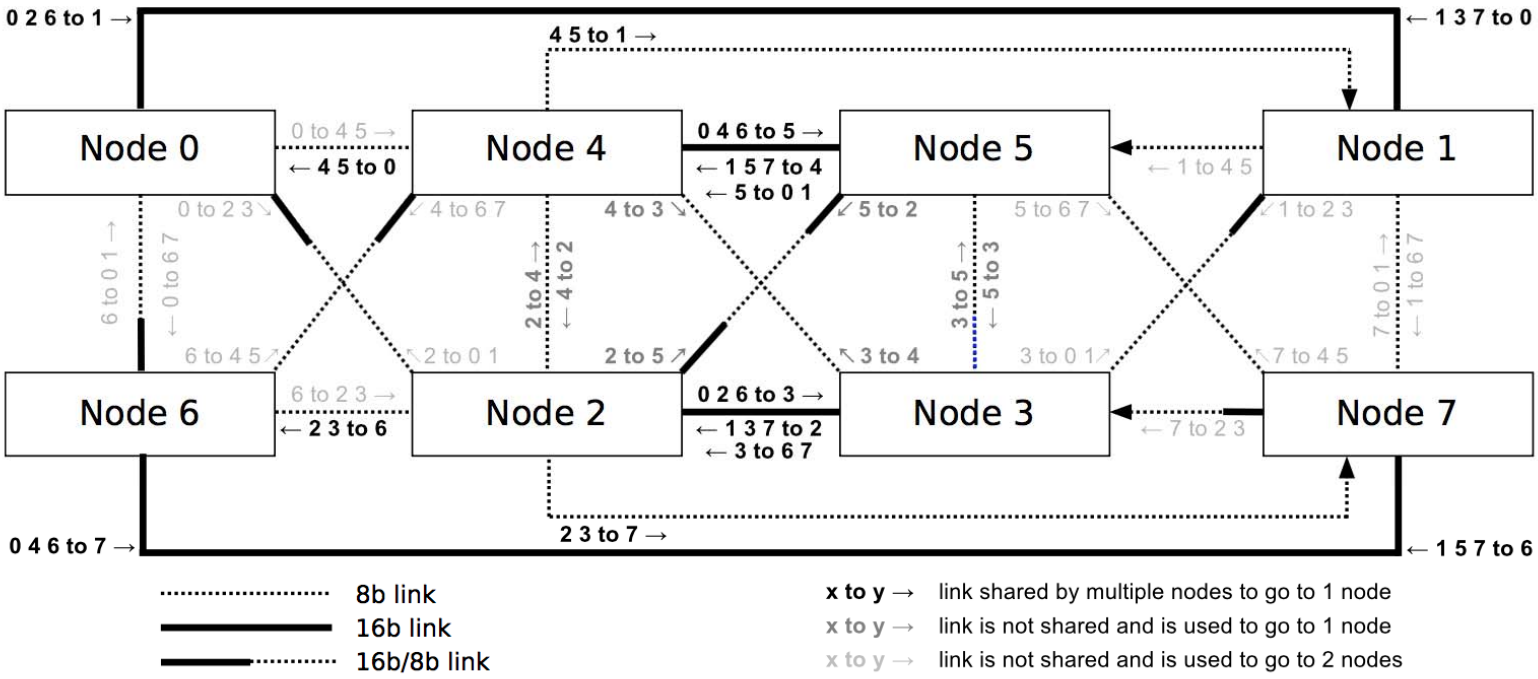


Hybrid NUMA



NORMA

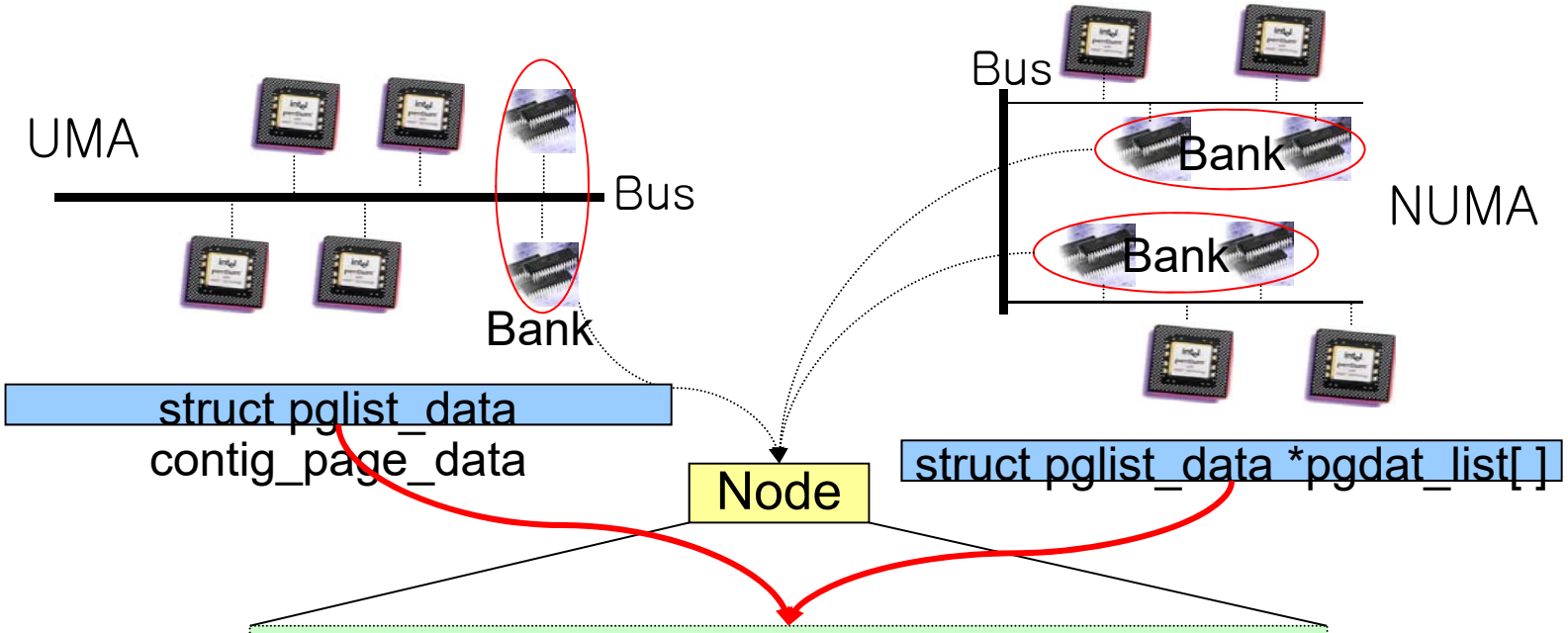
(Source : Unix Internals)



(Machines A and B)

Figure 1: Modern NUMA systems, with eight nodes. The width of links varies, some paths are unidirectional (e.g., between 7 and 3) and links may be shared by multiple nodes. Machine A has 64 cores (8 cores per node - not represented in the picture) and machine B has 48 cores (6 cores per node). Not shown in the picture: the links between nodes 4 and 1 and between nodes 2 and 7 are bidirectional on machine B. This changes the routing of requests from node 7 to 2 and node 1 to 4.

Bank and Node




```
typedef struct pglist_data {
        struct zone node_zones[MAX_NR_ZONES];
        struct zonelist node_zonelists[GFP_ZONETYPES];
        int nr_zones;
        struct page *node_mem_map;
        struct bootmem_data *bdata;
        unsigned long node_start_pfn;
        unsigned long node_present_pages;
        unsigned long node_spanned_pages;
        int node_id;
        wait_queue_head_t       kswapd_wait;
        wait_queue_head_t           pfmemalloc_wait;
        ...
} pg_data_t;
```

NODE 자료 구조

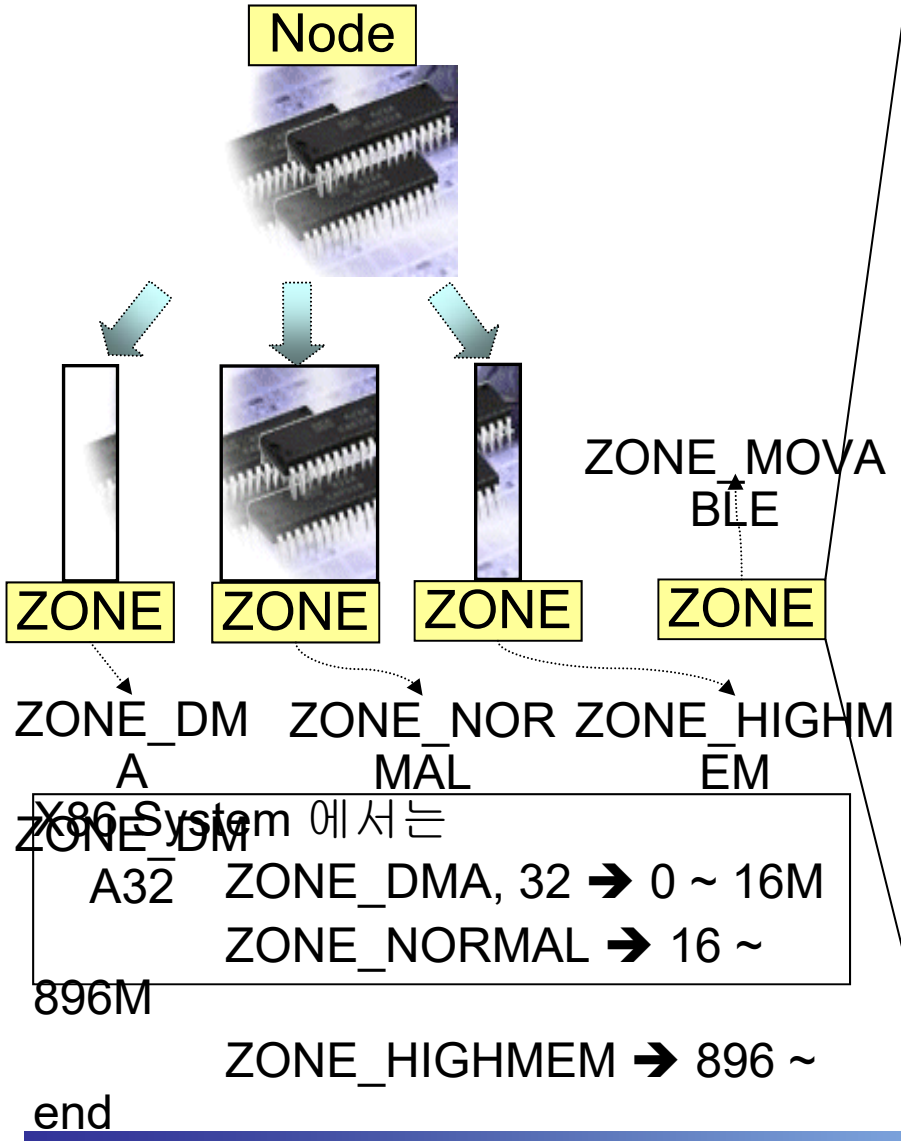
```
typedef struct pglist_data{
        zone_t node_zones[MAX_NR_ZONES];
                // node를 위한 zone은 ZONE_HIGHMEM, ZONE_NORMAL, ZONE_DMA 3개이다.
                // 각 zone을 가리키기 위한 배열임
        zonelist_t node_zonelists[GFP_ZONEMASK+1];
                // ((0x0f=15)+1=16) 할당 시에 우선시 되는 zone 순서를 나타냄
                // free_area_init_core()에의해
                // mm/page_alloc.c내의 build_zonelists()가 호출되면 이 순서를 정렬해 놓는다
        int nr_zones;
                // 이 node에 몇개의 zone이있는지 나타내는 1~3사이의 값.
                // 예를 들어 어떤 CPU는 ZONE_DMA 에 해당되는 메모리 영역이 없을수도 있으므로 항상 3은 아니다
        struct page *node_mem_map;
                // node의 각 physical frame을 나타내는 struct page배 열의 첫번째 page임.
                // 이는 전역배열인 mem_map의 어딘가에 들어갈 것이다
        unsigned long *valid_addr_bitmap;
                // memory node상에서, 실제로 존재하지않는 'holes'를 나타내기 위한 BITMAP
                // 사실상, Sparc과 Sparc64에서만 사용되며 다른 arch에선 무시됨
        struct bootmem_data *bdata
                // boot memory allocator가 사용하는 필드
        unsigned long node_start_paddr;
                // node의 physical 한 시작 주소
                // unsigned long은 PAE(physical Address Extension)을 사용하는 IA32나
                // PPC440GP같은 PowerPC의 변종들에선 최적화되어 작동하지 않는다
                // 좀더 나은 방법은 PFN(Page Frame Number)를 기록하는 것이다.
                // PFN은 간단히 말해서 page-size단위로 계산되는 physical memory의 index이다
                // PFN은 보통 (page_phys_addr >> PAGE_SHIFT)로 정의 된다.
        unsigned long node_start_mapnr;
                // mem_map내에서의 page offset을 나타냄
                // 이 숫자는 ,mem_map과 lmem_map이라고 불리는 local mem_map사이의 page갯수를 계산하는,
                // free_area_init_core()에서 계산됨
        unsigned long node_size;
                // 이 node내의 총 page수
        int node_id;
                // 0에서 시작되는 Node ID(NID)
        struct pglist_data *node_next;
                // NULL로 끝나도록 되어있는, 다음 node를 가리키는 pointer
}pg_data_t;
```

NODE 자료 구조 확인


```
contig_page_data = (
  node_zones = (
    (
      lock = (),
      free_pages = 11510,
      pages_min = 128,
      pages_low = 256,
      pages_high = 384,
      need_balance = 0,
      free_area = ((free_list = (next = 0xC021ED04, prev = 0xC021E204), map = 0xC02B1000), (free_lis
      wait_table = 0xC0003000,
      wait_table_size = 64,
      wait_table_shift = 26,
      zone_pgdat = 0xC0198330,
      zone_mem_map = 0xC0200014,
      zone_start_paddr = 2684354560,
      zone_start_mapnr = 0,
      name = 0xC016F320,
      size = 16384),
    (lock = (), free_pages = 0, pages_min = 0, pages_low = 0, pages_high = 0, need_balance = 0, free
    (lock = (), free_pages = 0, pages_min = 0, pages_low = 0, pages_high = 0, need_balance = 0, free
  node_zonelists = ((zones = (0xC0198330, 0x0, 0x0, 0x0)), (zones = (0xC0198330, 0x0, 0x0, 0x0)), (z
  nr_zones = 1,
  node_mem_map = 0xC0200014,
  valid_addr_bitmap = 0x0,
  bdata = 0xC01D4A48,
  node_start_paddr = 2684354560,
  node_start_mapnr = 0,
  node_size = 16384,
  node_id = 0,
  node_next = 0x0)
```

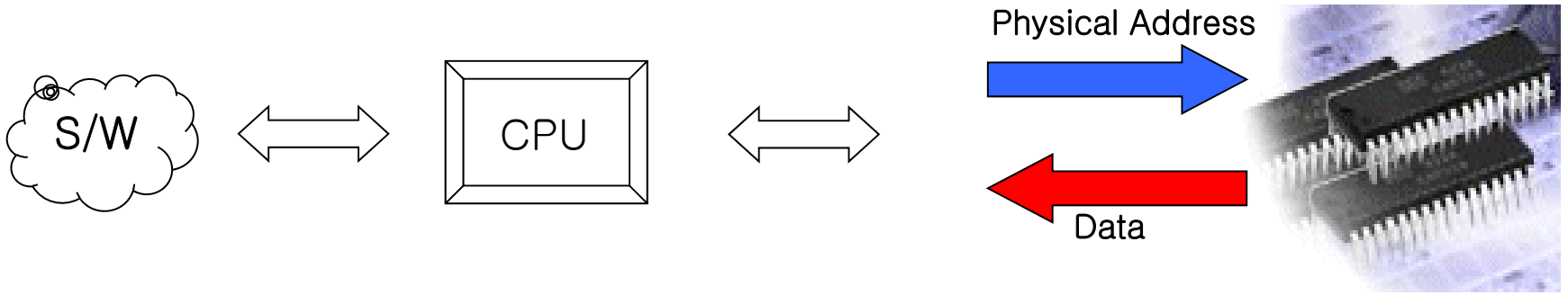
☞**0xa0000000** ➔ 실험을 진행하고 있는 보드의 물리메모리 시작 주소

Node and Zone

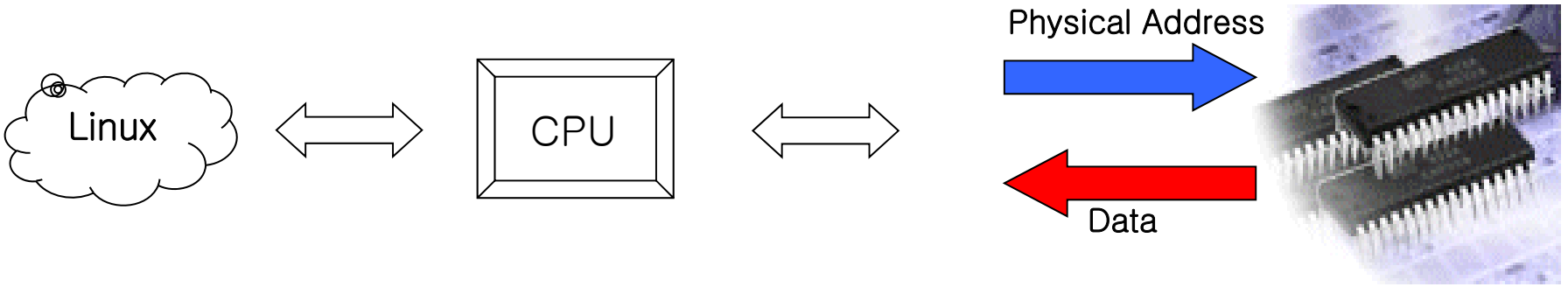



```
struct zone {
        unsigned long           watermark[NR_WMARK];
        unsigned long           lowmem_reserve[MAX_NR_ZONES];
        unsigned long           dirty_balance_reserve;
        int                     node;
        unsigned long           min_unmapped_pages;
        unsigned long           min_slab_pages;
        spinlock_t              lock;
        struct free_area        free_area[MAX_ORDER];
        spinlock_t              lru_lock;
        struct lruvec           lruvec;
        atomic_long_t           inactive_age;
        unsigned long           pages_scanned;
        unsigned long           flags;
        atomic_long_t           vm_stat[NR_VM_ZONE_STAT_ITEMS];
        unsigned int            inactive_ratio;
        wait_queue_head_t       * wait_table;
        unsigned long           wait_table_hash_nr_entries;
        unsigned long           wait_table_bits;
        struct per_cpu_pageset  *pageset;
        struct pglist_data      *zone_pgdat;
        struct page             *zone_mem_map;
        unsigned long           zone_start_pfn;
        char                    *name;
        unsigned long           spanned_pages;
        unsigned long           present_pages;
}
```

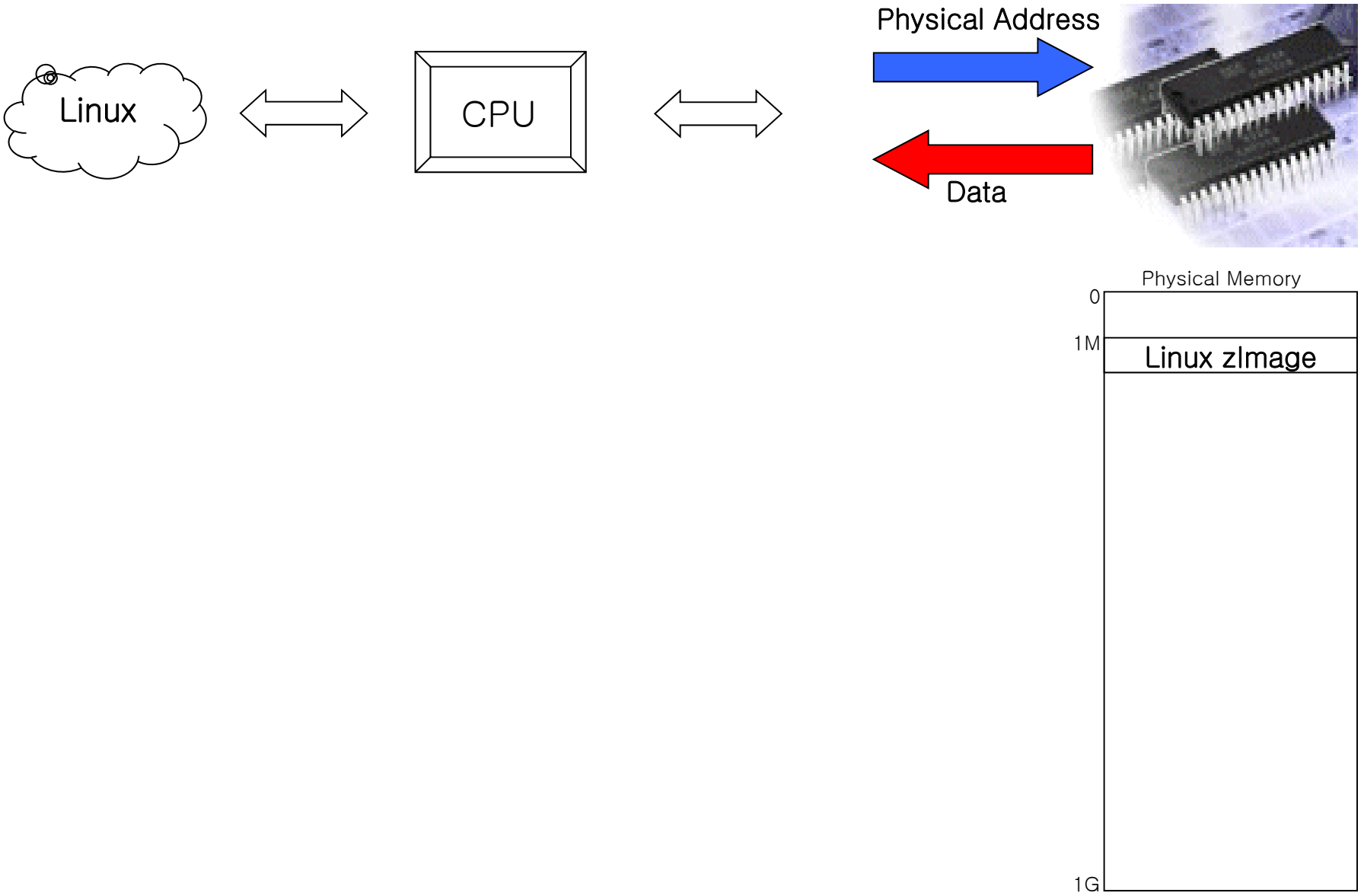
Kernel Address Space(1/13)



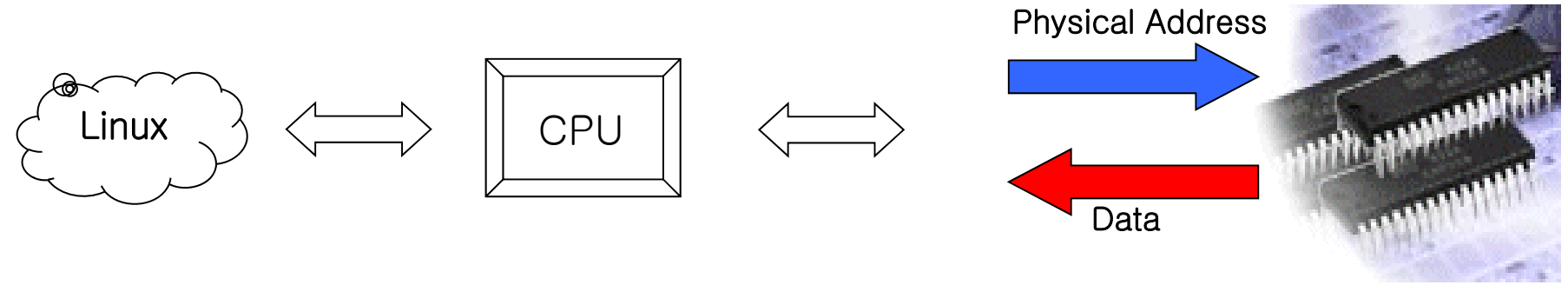
Kernel Address Space(2/13)

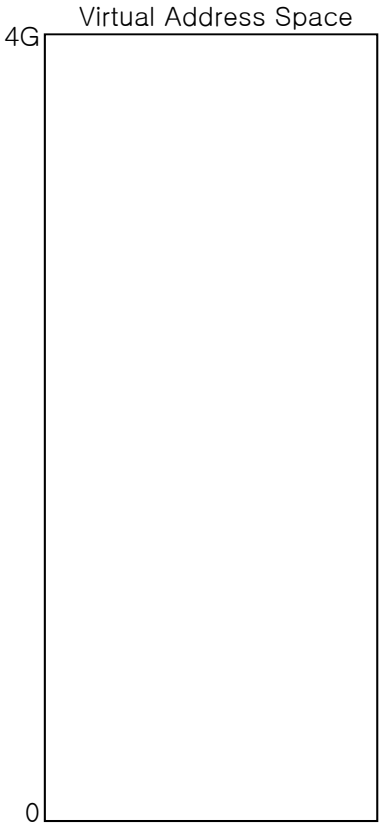


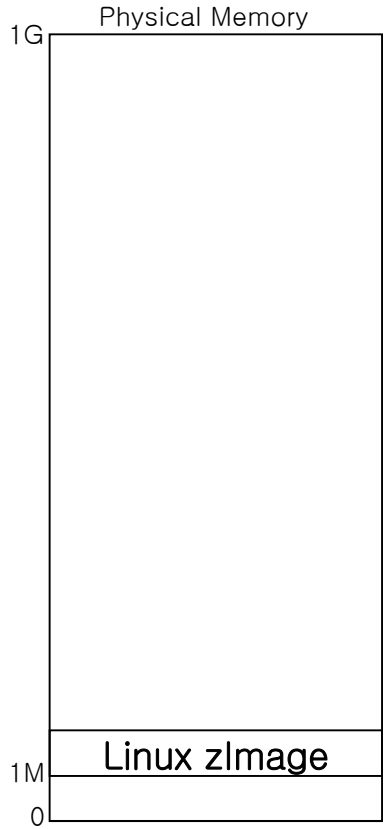
Kernel Address Space(3/13)



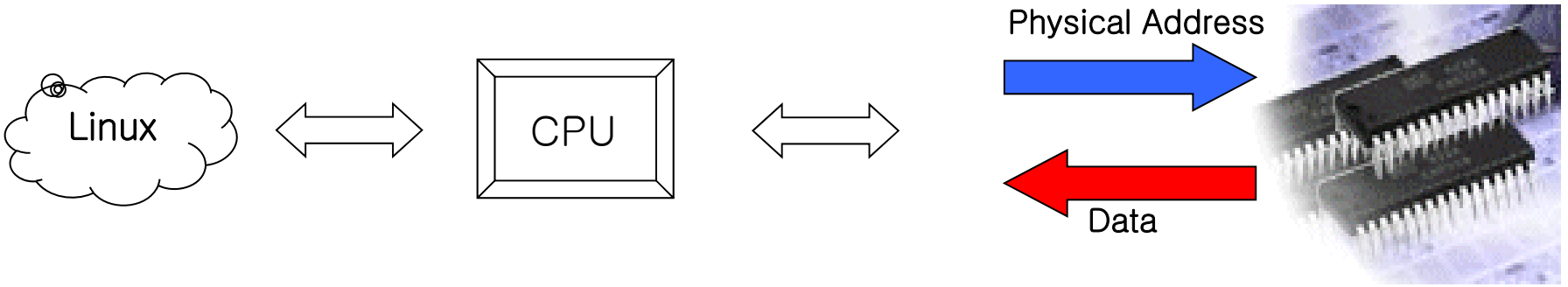
Kernel Address Space(4/13)

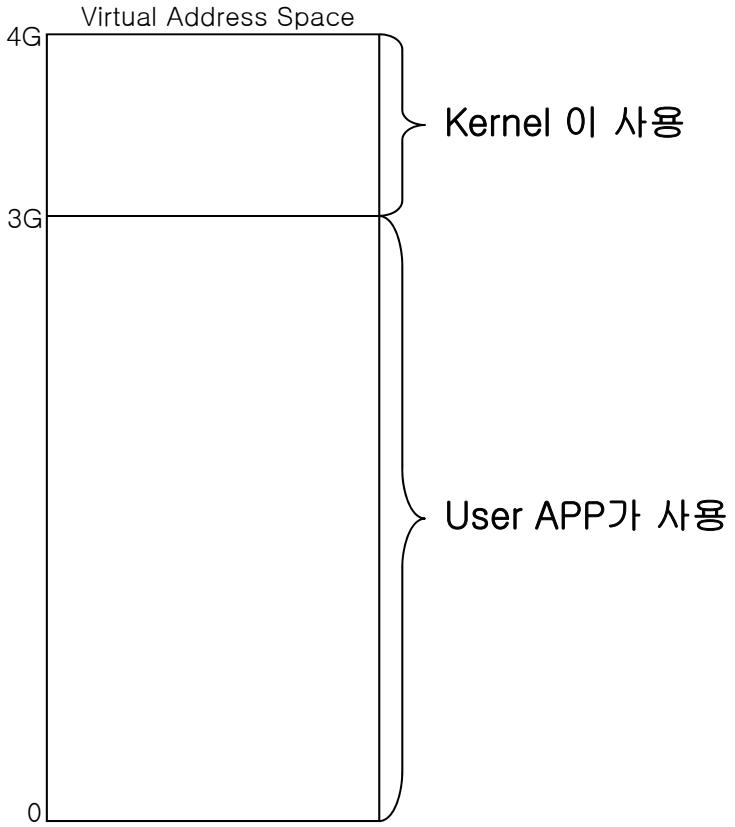


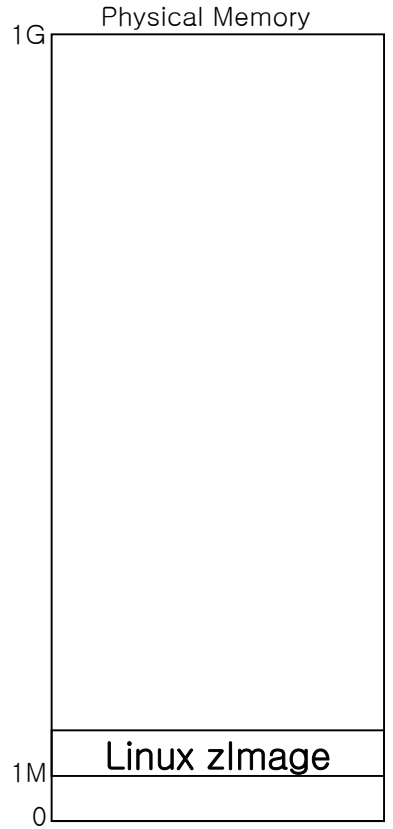




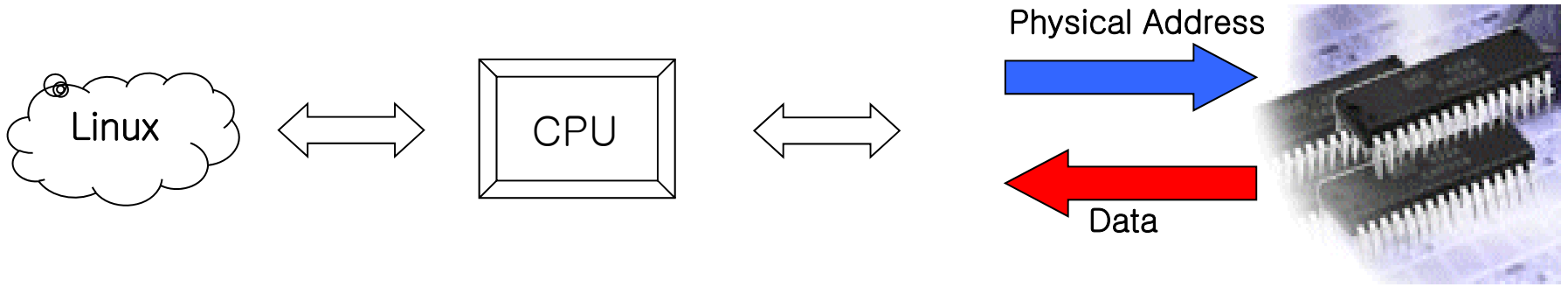
Kernel Address Space(5/13)



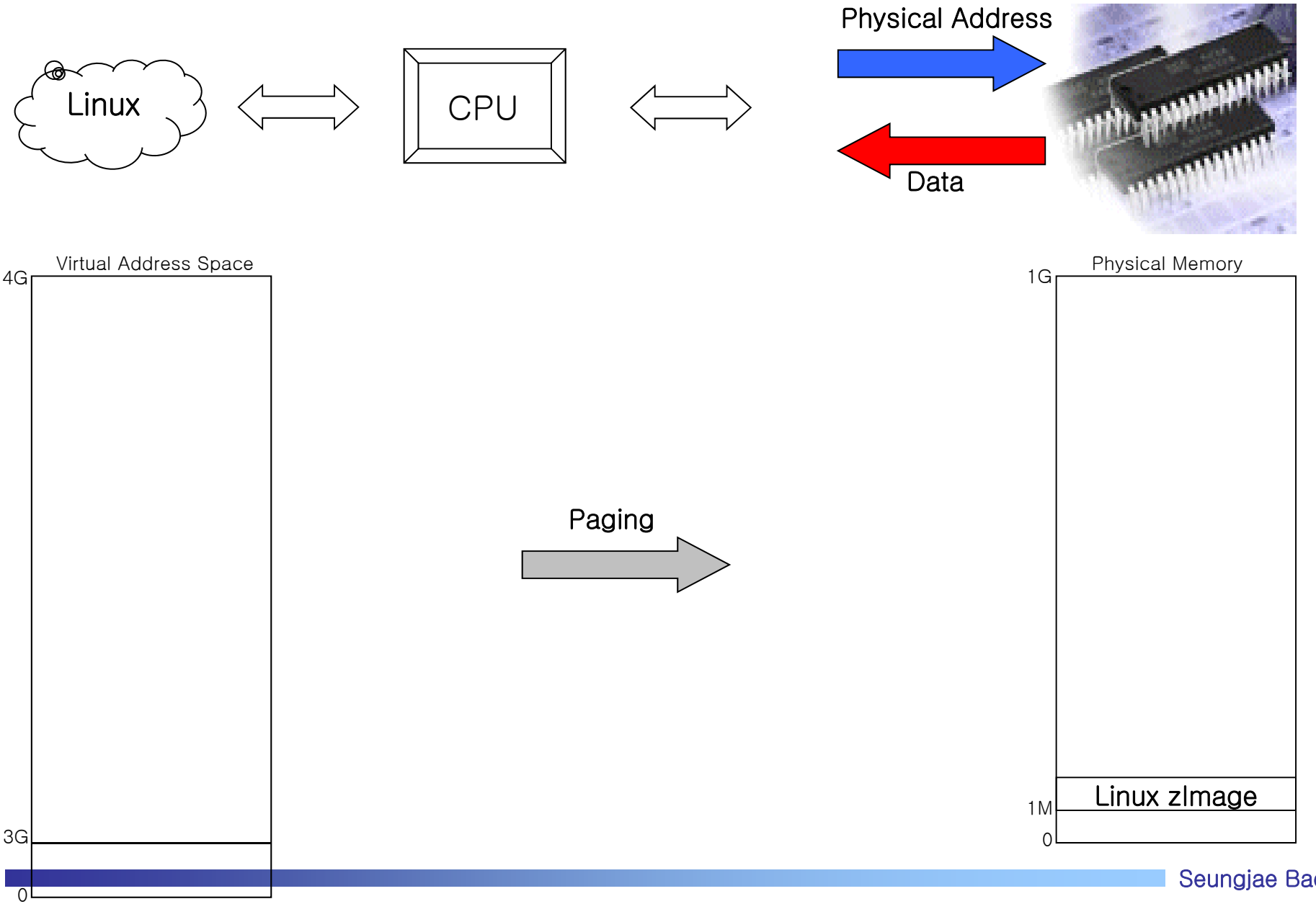




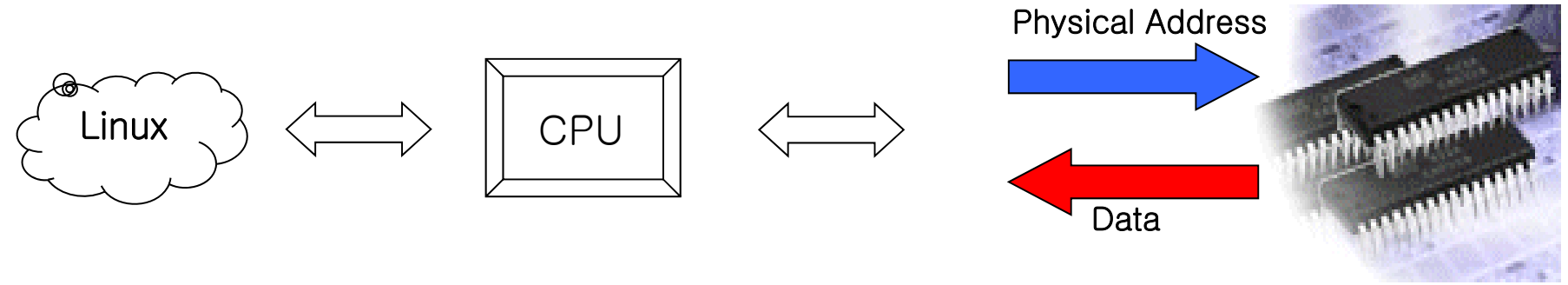
Kernel Address Space(6/13)

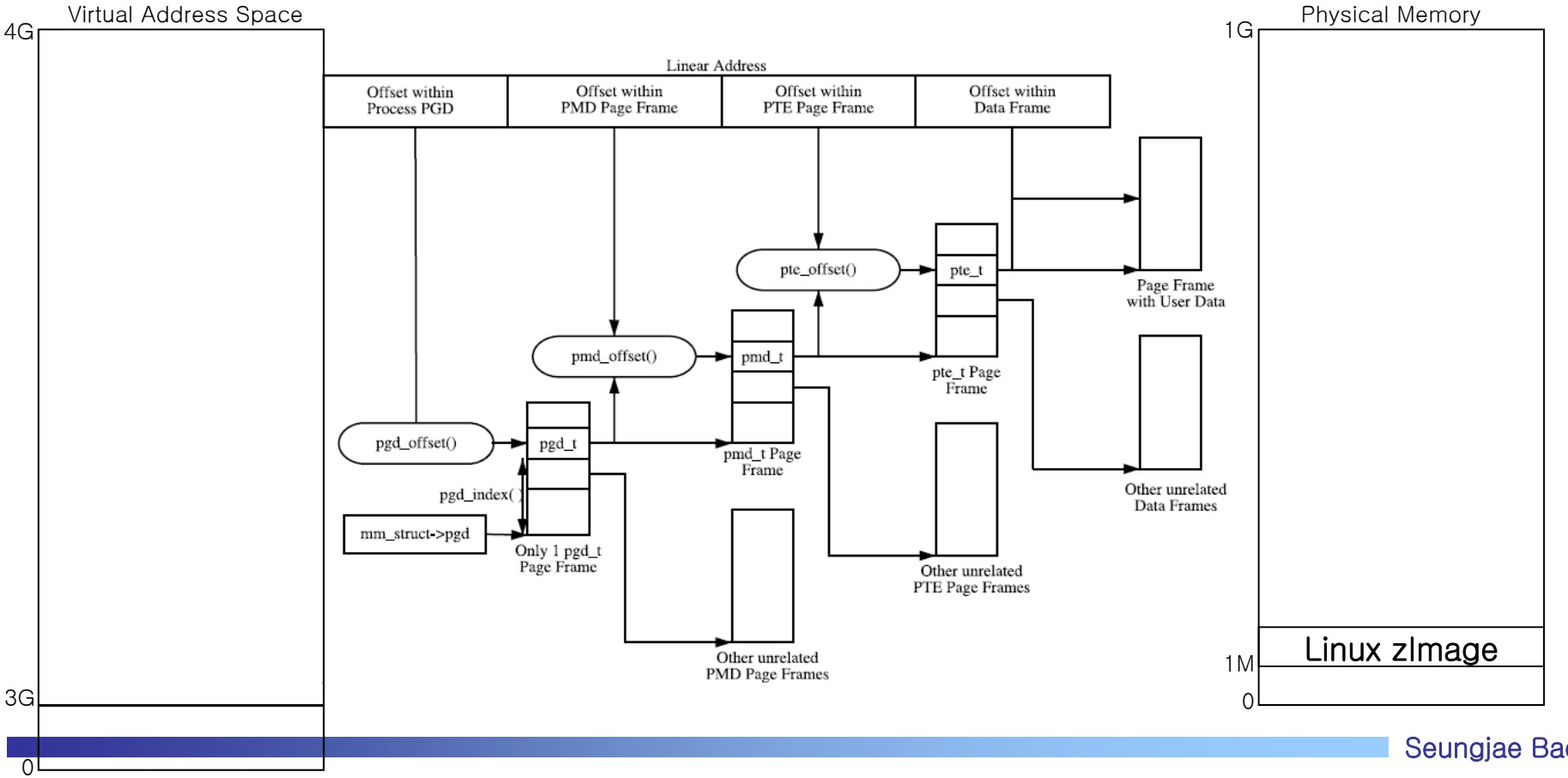


Kernel Address Space(7/13)

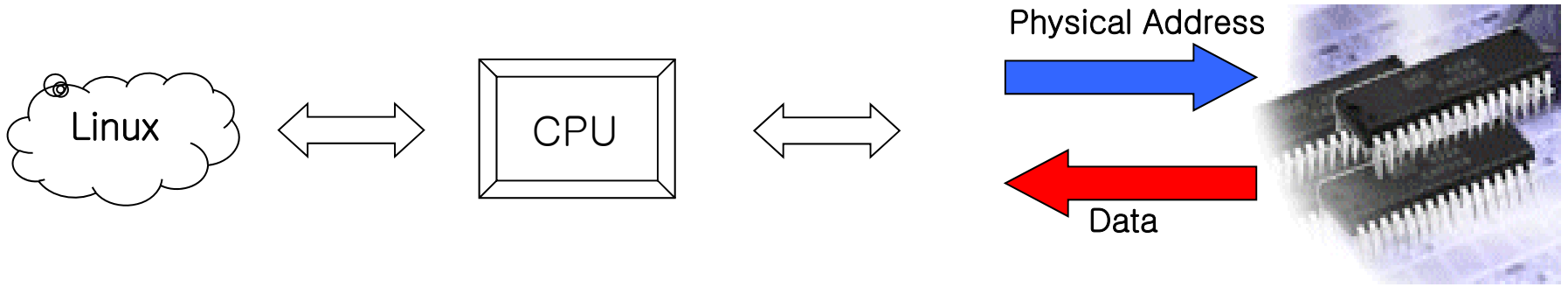


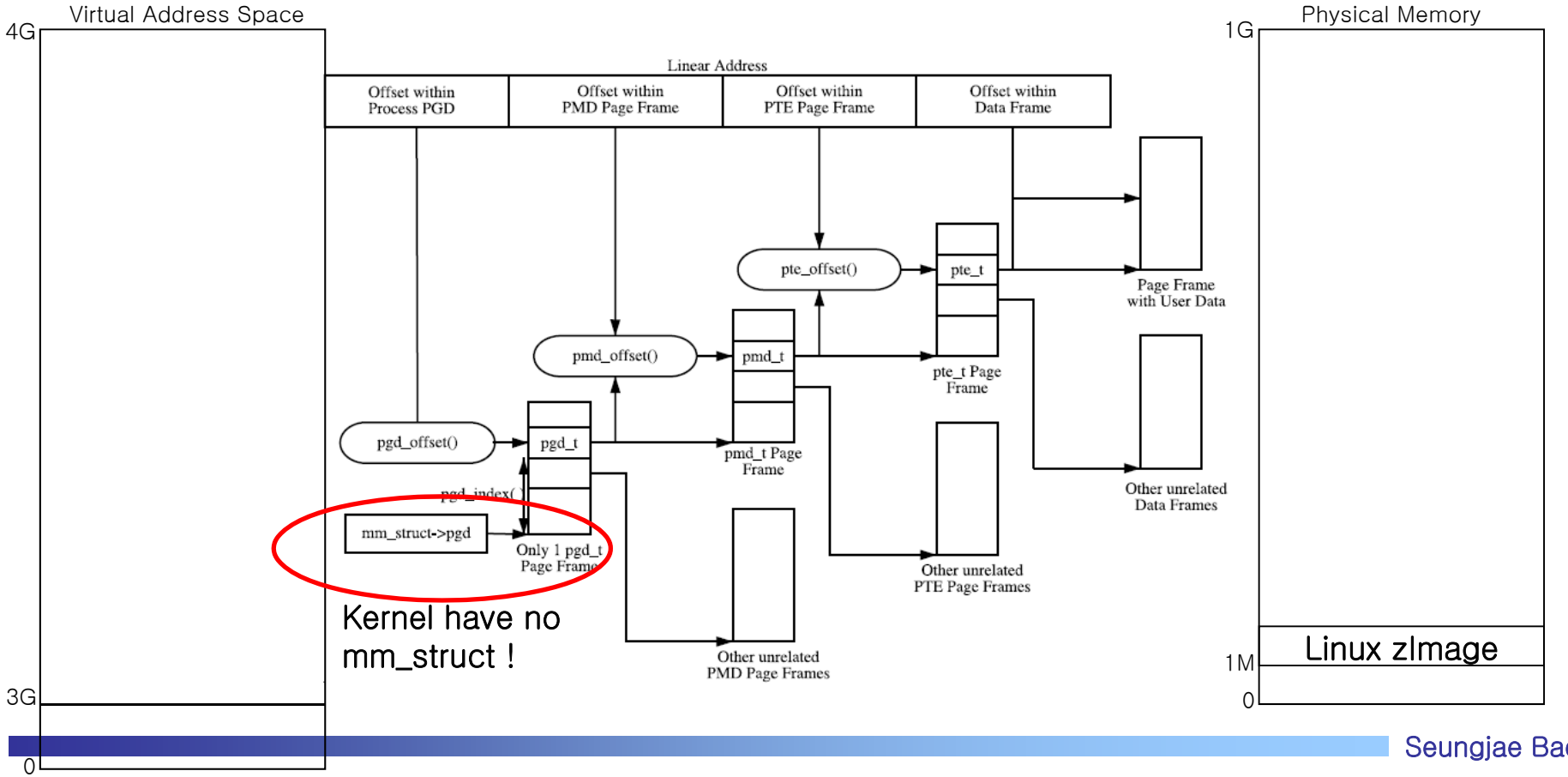
Kernel Address Space(8/13)



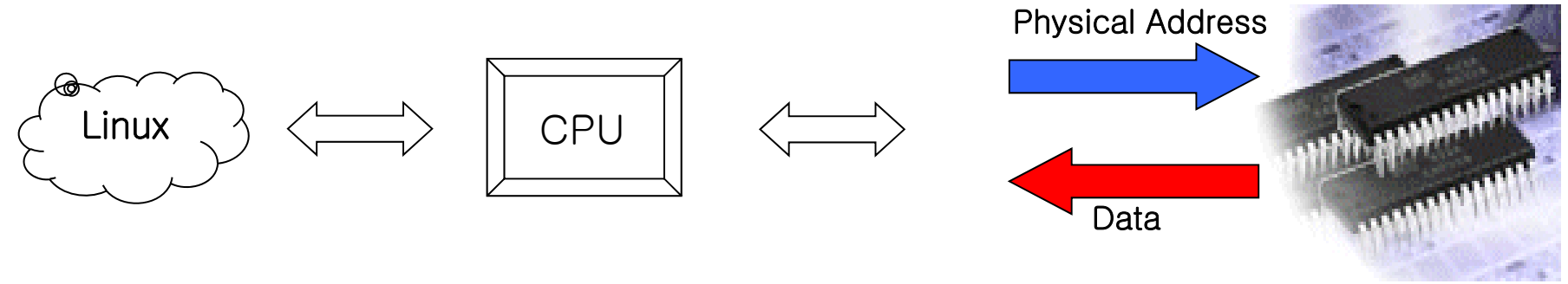


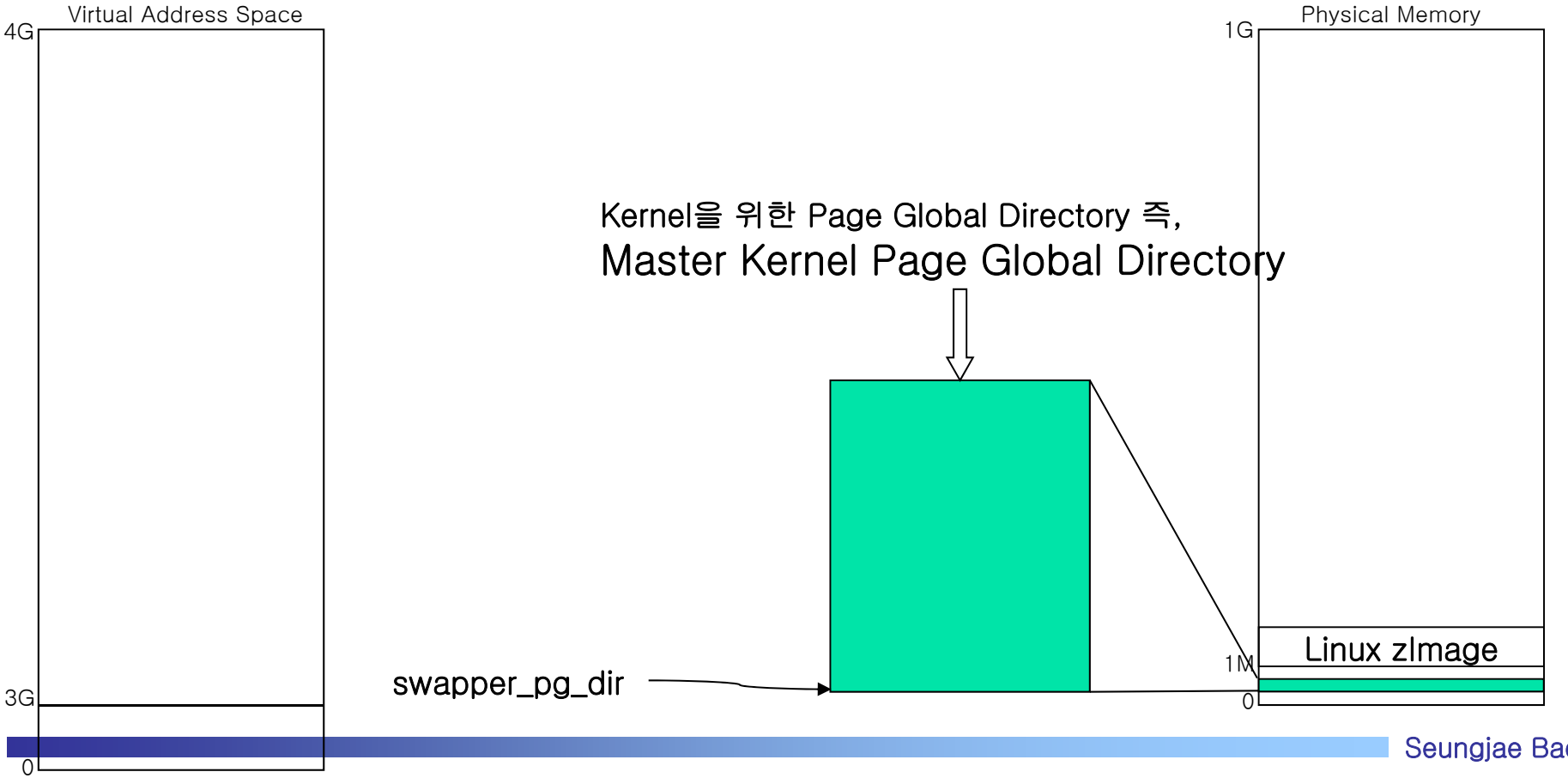
Kernel Address Space(9/13)



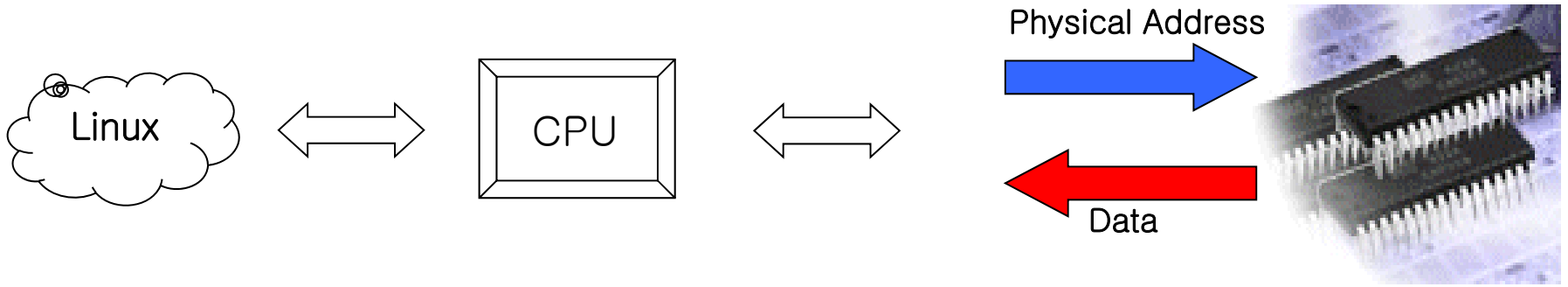


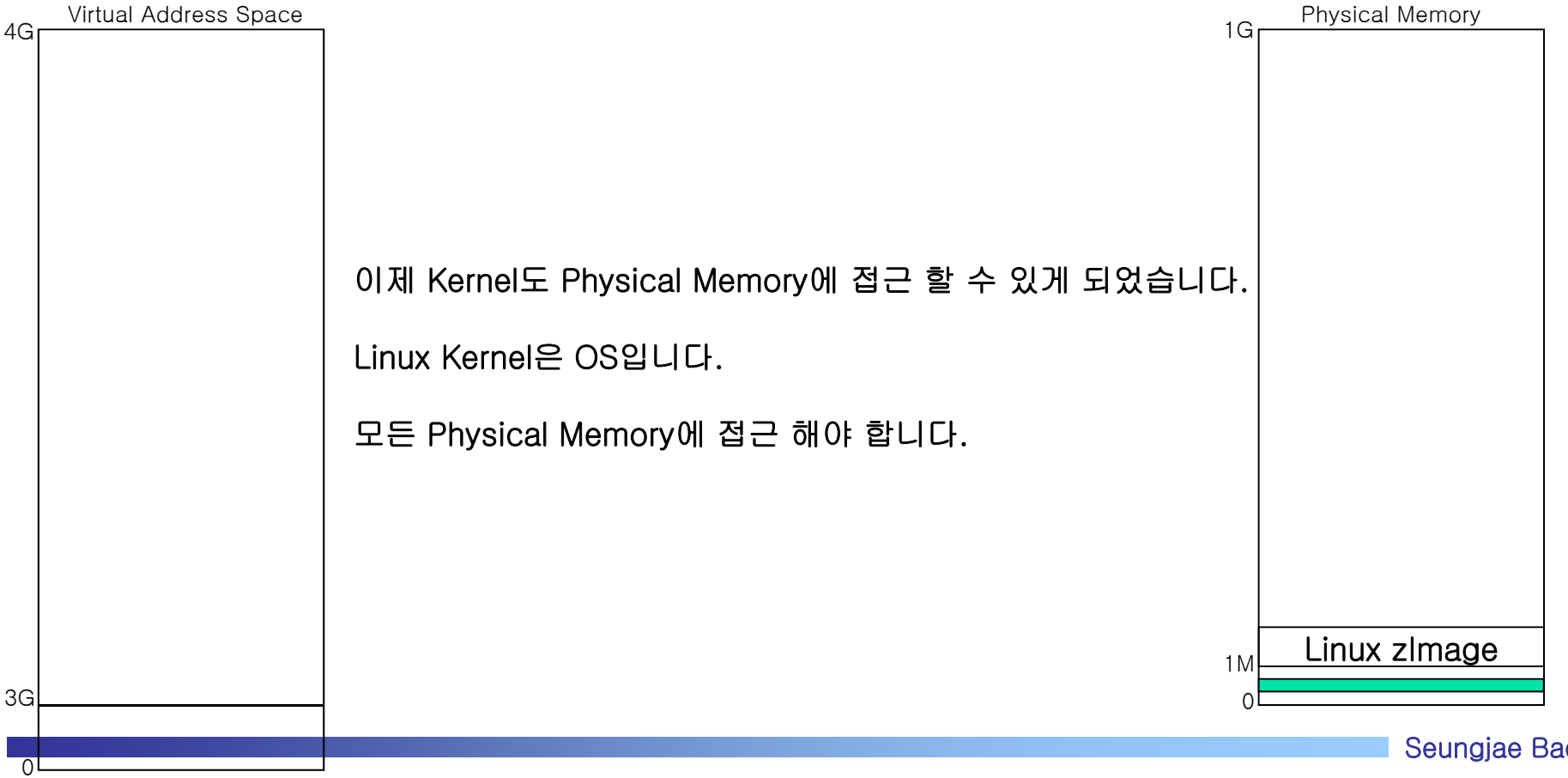
Kernel Address Space(10/13)





Kernel Address Space(11/13)



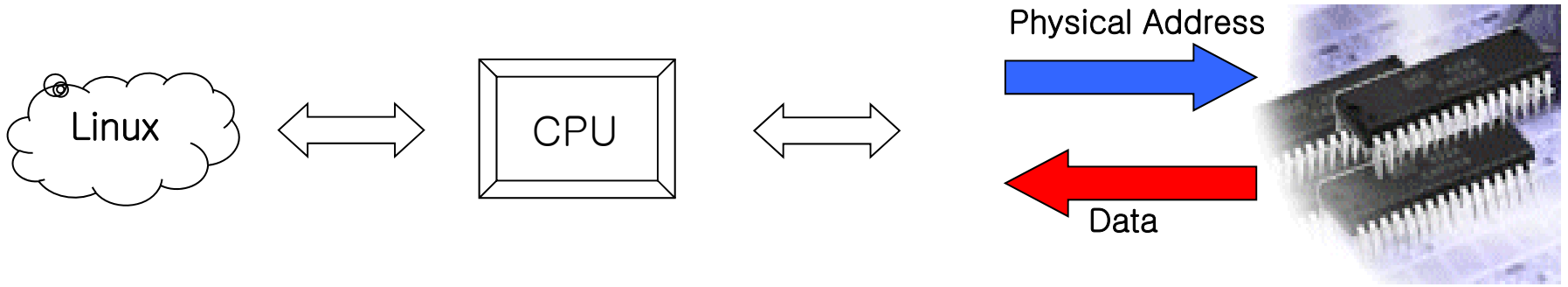


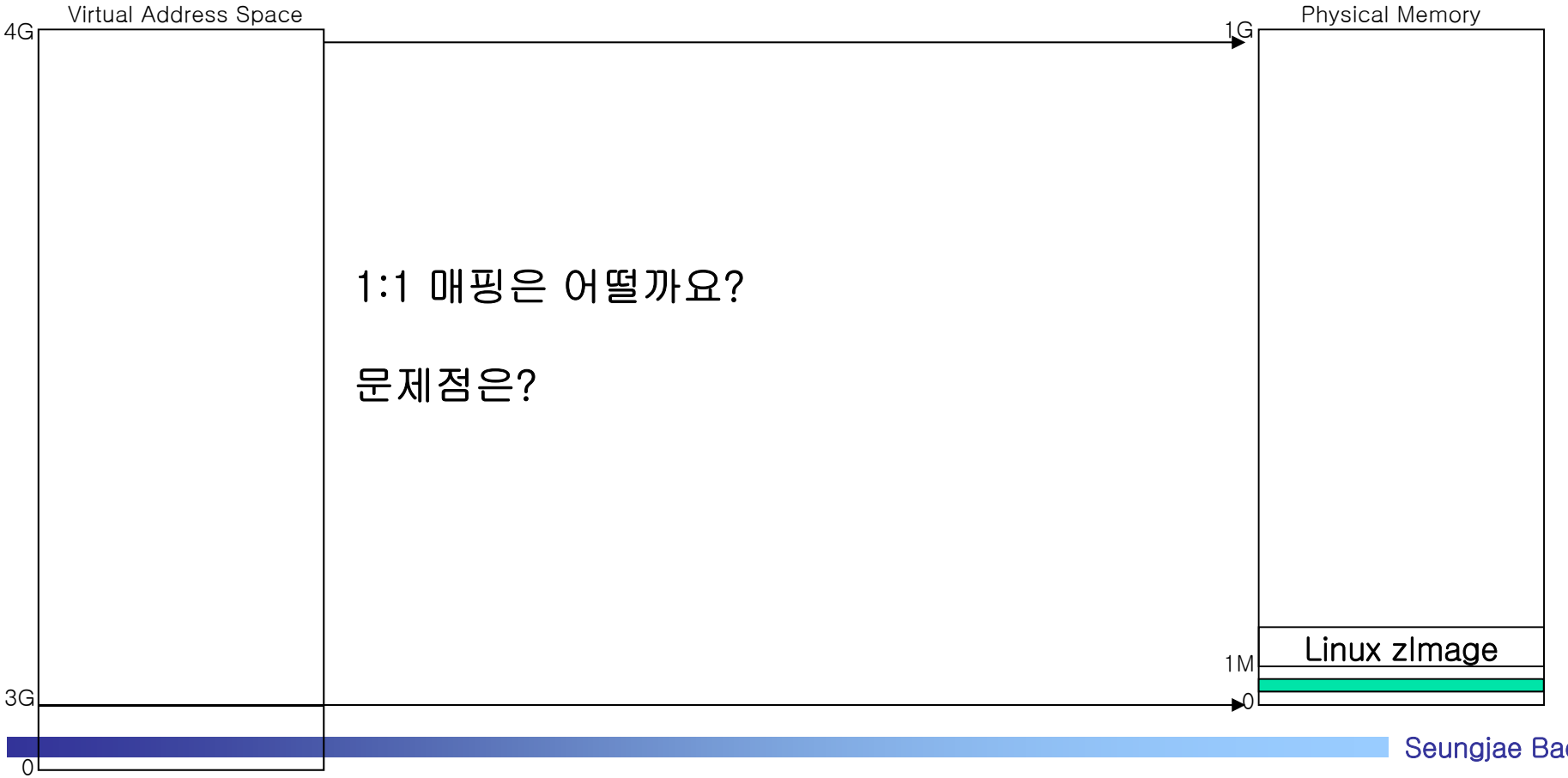
이제 Kernel도 Physical Memory에 접근 할 수 있게 되었습니다.

Linux Kernel은 OS입니다.

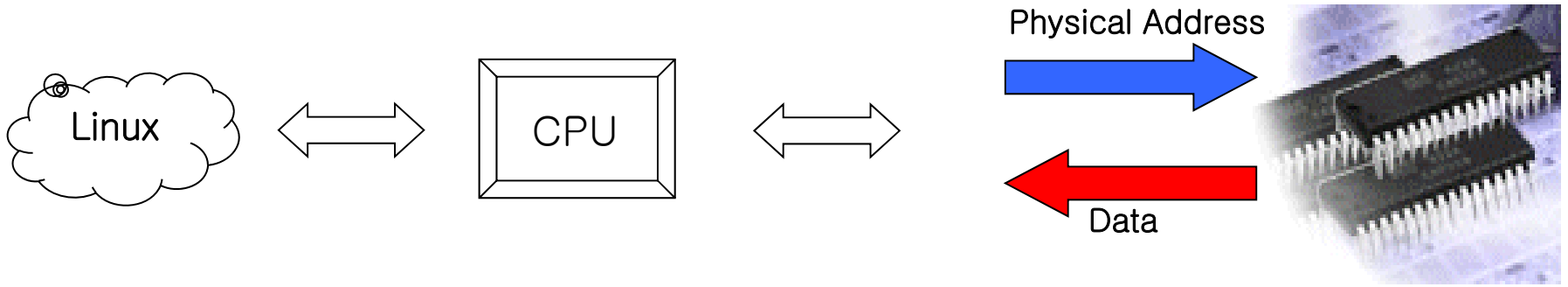
모든 Physical Memory에 접근 해야 합니다.

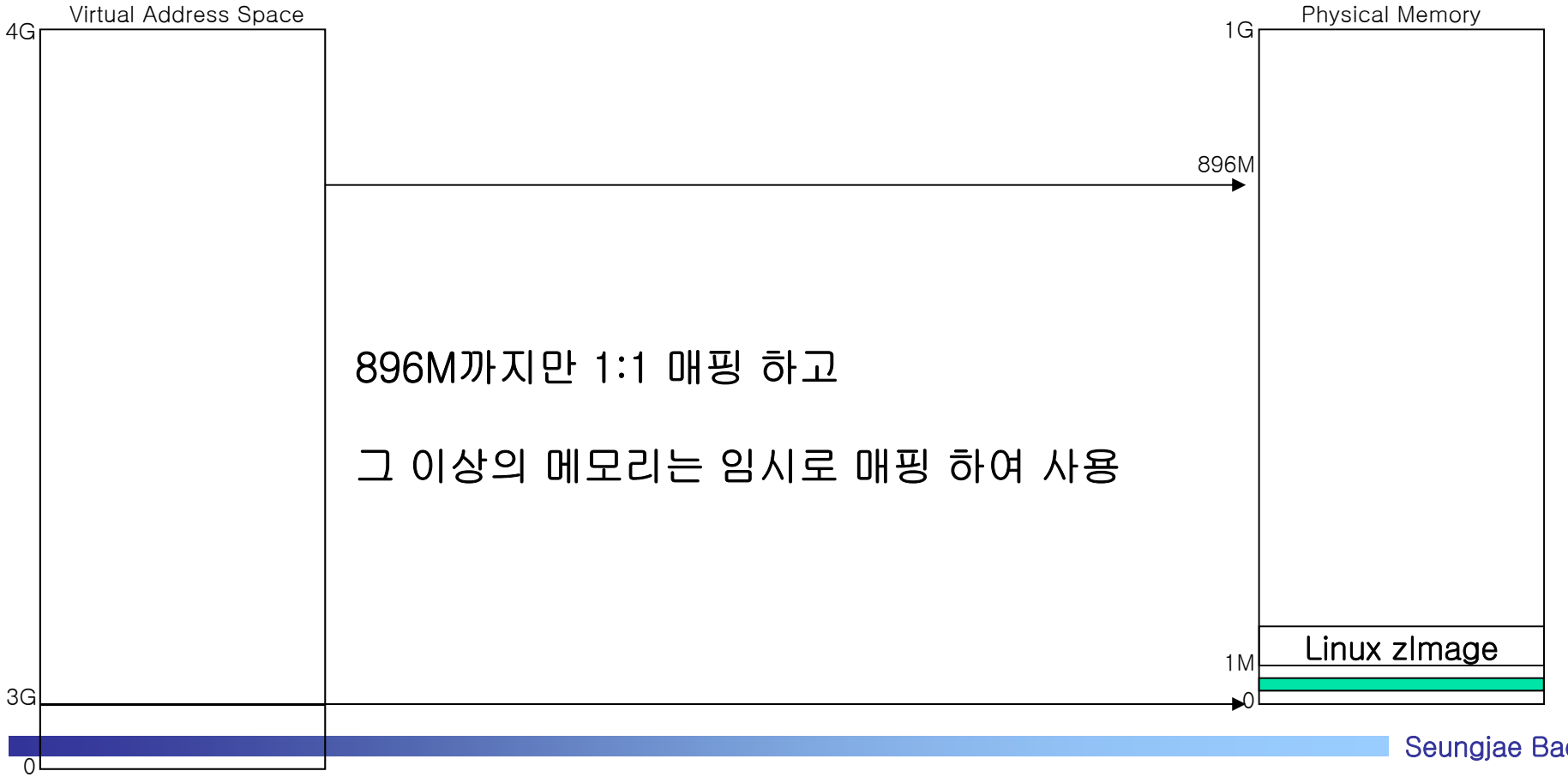
Kernel Address Space(12/13)





Kernel Address Space(13/13)



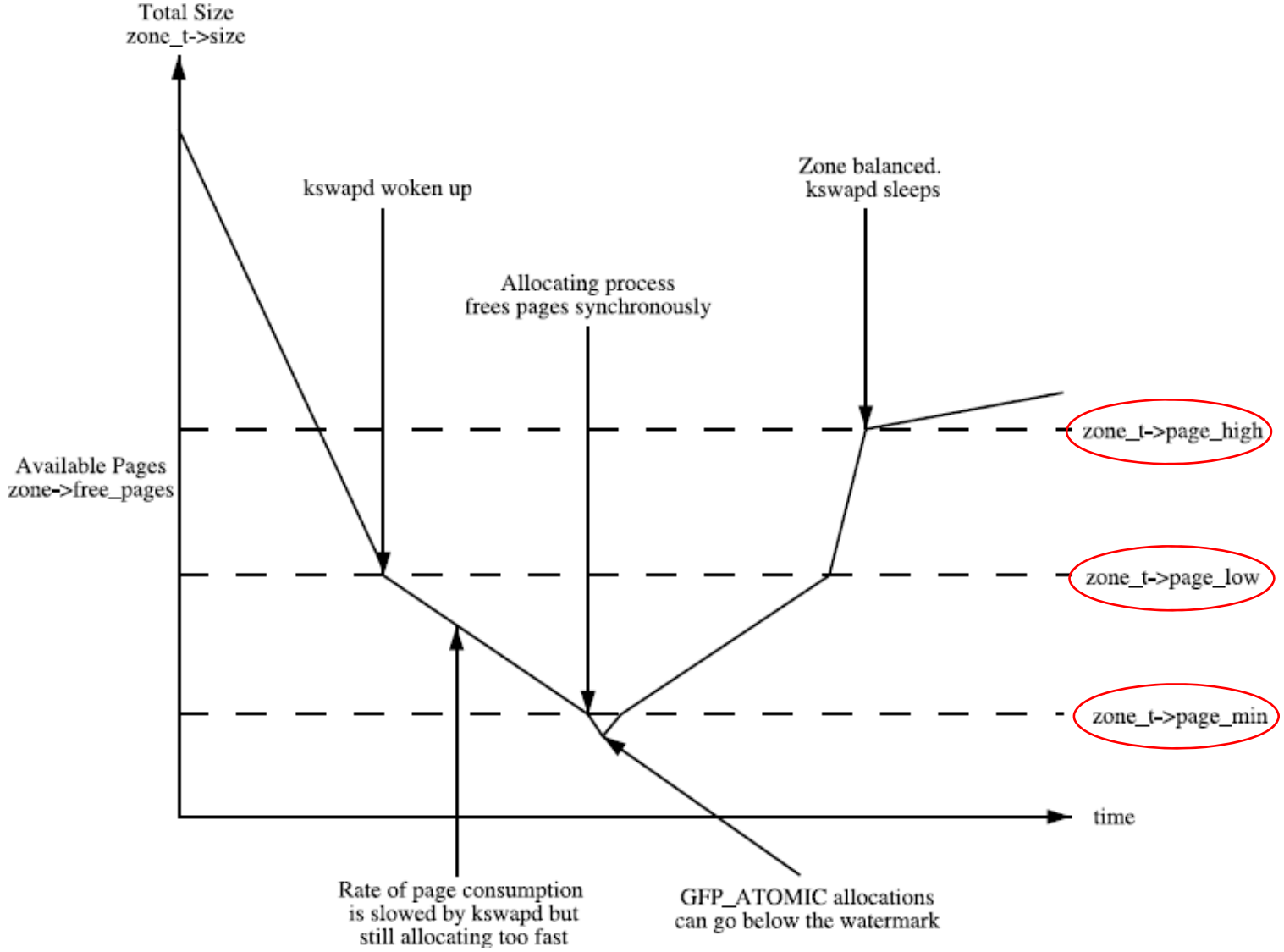


ZONE 자료 구조

```
typedef struct zone_struct{
                spinlock_t      lock;
                                //concurrent 한 접근 으로부터 zone을 보호하는 spinlock
                unsigned long   free_pages;
                                //현재 zone내의 free page들의 총 개수
                unsigned long   pages_min, pages_los, pages_high;
                                //zone watermarks
                int             need_balance;
                                //이 플래그는 zone의 균형을 유지하기 위해 kswapd 에게 pageout요청을 하는데 사용되는 flag이다
                free_area_t     free_area[MAR_ORDER];
                                //buddy allocator이 사용하는 free area bitmaps
                wait_queue_head_t *wait_table;
                                // Process들이 page가 free되기를 기다릴 때 사용되는 wait queues의 hash table이다
                                // 이는 wait_on_page()와 unlock_page()에서 매우 중요하다
                                // 물론 process들은 한 개의 queue에서 대기 할 수도 있지만, 이렇게 하게 된다면
                                // wake up 하게 되는 때에, 모든 process들은 lock걸리기 전까지는 page를 얻기 위해 경쟁하게 된다
                                // 이렇게 공유자원을 얻기 위해 다투는 프로세스들의 큰 그룹을
                                // Thundering herd라고 부른다.
                                // (thundering herd : 여러 프로세스가 깨어나서, 이중, 하나만 사용할 수  있는 자원을 차지하려고 경쟁하고,
                                //                    나머지 프로세스는 다시 잠든 상태로 돌아가는 상황)
                unsigned long   wait_table_size;
                                //hash table내의 queue개수를 결정하는, 2의 제곱의 수
                unsigned long   wait_table_shift;
                                //위에 정의된 table size를 계산하는데 사용되는 이진 비트 조합
                struct pglist_data *zone_pgdat;
                                //부모 pg_data_t를 가리키는 pointer
                struct page     *zone_mem_map;
                                //현재 zone이 참조하게 되는 전역 배열 mem_map내의 첫 번째 page
                unsigned long   zone_start_paddr;
                                //node_start_paddr과 유사하게 사용됨
                unsigned long   zone_start_mapnr;
                                //node_start_mapnr과 유사하게 사용됨
                char            *name;
                                //ZONE을 나타내는 'DMA', 'NORMAL', 'HIGHMEM'중 하나의 문자열
                unsigned long   size;
                                // page단위로 표현되는 zone의 크기
}zone_t;
```

ZONE의 watermarks

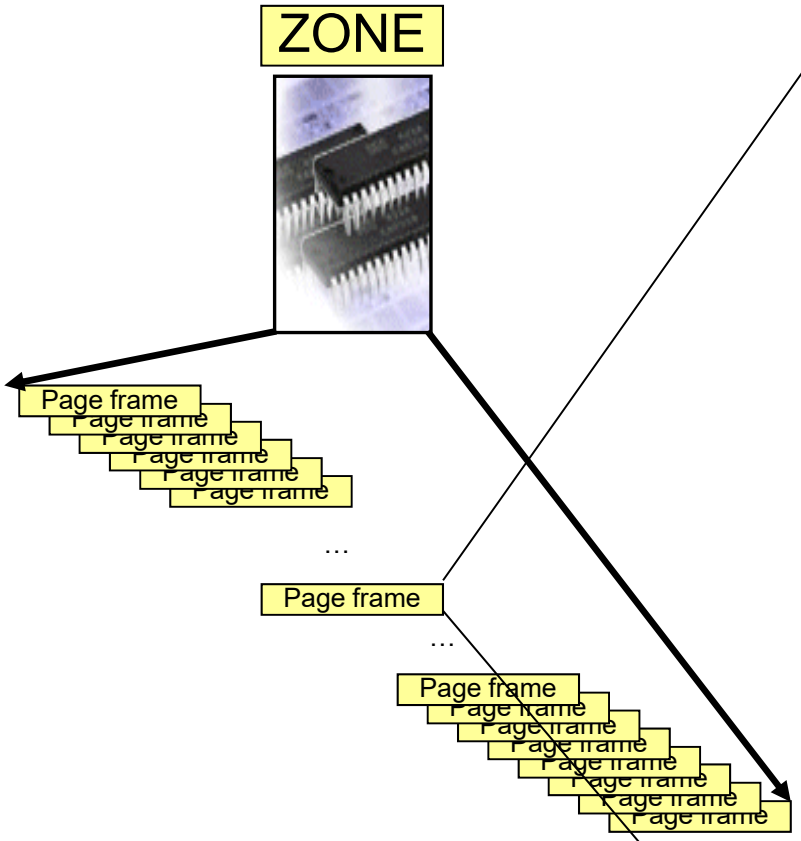
- Zone Watermarks



Watermarks 계산

- Zone Watermarks
  - ✓ page_low, pages_min, pages_high
  - ✓ 보통
    - pages_low = pages_min * 2
    - pages_high = pages_min * 3
  - ✓ pages_min 필드는
    - free_area_init_core() 함수 내에서 계산됨
    - 보통 (ZoneSizeInPages / 128)

ZONE과 PageFrame




```
struct page {
        page_flags_t flags;
        union {
              struct address_space *mapping;
              void *s_mem;
        }
        struct {
              union {
                    pgoff_t index;
                    void *freelist;
                    bool pfmemalloc
              };
              union{
                    ...
              }
        }
        union {
              struct list_head lru;
              struct {
                    struct page *next;
                     int pages; pobjects;
              }
              struct slab *slab_page;
              struct rcu_head rcu_head;
        }
        ...
#if defined(WANT_PAGE_VIRTUAL)
        void *virtual;
#endif /* WANT_PAGE_VIRTUAL */
};
```

각 NODE의 모든 page 구조체는 보통 ZONE_NORMAL의 시작부분
(혹은 커널이 올라 오기 위해 예약해 놓은 메모리 바로 다음)
위치에 있는 전역 배열인 "**mem_map**"에 유지된다

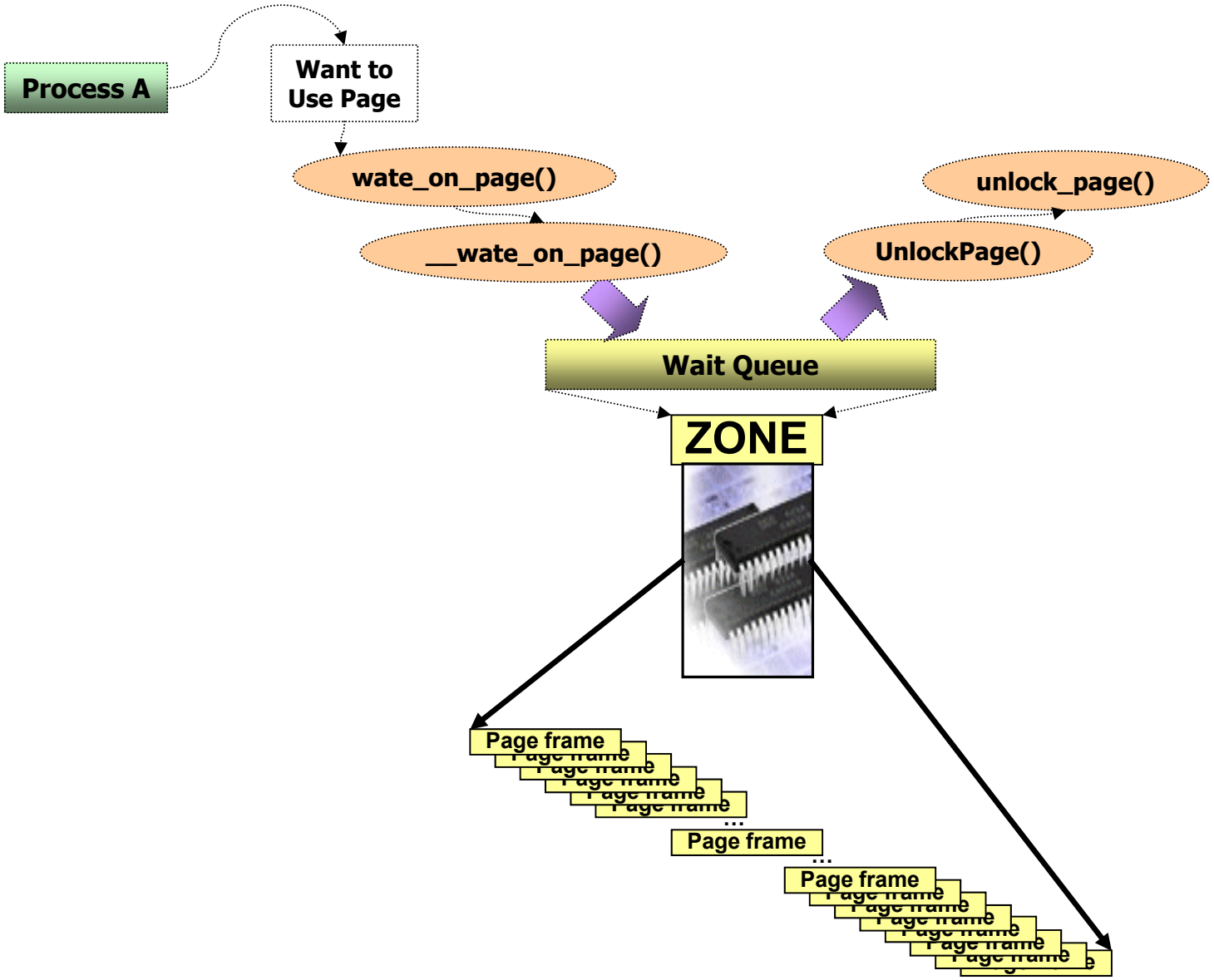
Page Frame 자료 구조

```
typedef struct page{
    struct list_head list;
        // page는 많은 list에 속해 있을 수 있고, 이 필드는 list head를 위해 사용된다.
        // 예를들어, mapping되어 있는 page는 address_space에 의해 관리되는 세 개의 원형 링크드 리스트 중 하나에 속해 있을 것이다.
        // 이 세개의 원형 링크드 리스트는 clean_pages, dirty_pages, locked_pages이다.
        // slab allocator에서 이필드는 슬랩 할당자에 의해 할당되었을때 page를 관리하기 위해
        // slab이나 cache structures로의 포인터를 저장하는 역할을 한다.
        // 또한 free pages의 link blocks 에서도 사용된다.
    struct address_space *mapping
        // 파일이나 device가 memory mapped되어 있을 때, 그들의 inode는 address_space와 연결되어 있다.
        // 만약 이 page가 파일에 연결되어 있다면 이 필드는 address space를 가리킨다.
        // 만약 page가 anonymous이며 mapping이 set되어 있다면,
        // address_space는 swap address space를 관리하는 swapper_space이다.
    unsigned long index;
        // 이 필드는 두가지 용도를 가지며, 이는 page의 state가 결정한다.
        // 만약 이 page가 file mapping의 일부라면 이는 file내의 offset이다
        // 만약 이 page가 swap cache의 일부라면 이는 swap address space(swapper_space)를 위한 address_space내의 offset이 됨
        // 둘째, 만약 page들의 블록이 특정 process를 위해 free되었다면
        // 블록 내에서의 순서가 저장될 것이다. 이는 __free_pages_ok()함수 내에서 set 된다
     struct page *next_hash;
        // 파일 매핑의 일부인 page는 inode와 offset에 의해 hash된다.
        // 이 필드는 같은 hash bucket을 공유하는 페이지 들을 link시켜주는 필드이다.
```

Page Frame 자료 구조

```
    atomic_t count;
        // page의 참조 횟수이다. 만약 0이 되면 이 페이지는 free될 것이다.
        // 그보다 크다면, 하나 이상의 프로세스에서 사용 중 이거나,
        // I/O를 기다리기 위한 작업등으로 인해 커널 내에서 사용 중임을 나타낸다.
    unsigned long flags;
        // page의 상태를 나타내는 flags이다. 이는 <linux/mm.h>내에 모두 정의 되어 있고, 표 2.1에 나열해 놓았다.
        // bit를 test, clear, set하기 위한 여러 개의 매크로가 정의 되어 있으며, 이를 표 2.2에 보였다.
        // 사실 유일하게 관심 있는 함수는 아키텍쳐에 의존적인 함수인 arch_set_page_uptodate()를 call하는SetPageUptodate()이다.
    struct list_head lru;
        // page 교체 정책을 위해, 교체 되어 나갈 page는
        // page_alloc.c내에 정의 되어 있는 active_list나 inactive_list 둘 중 하나에 존재해야 한다.
        // 이 필드는 이러한 LRU리스트의 list head를 가리키게 된다.
    struct page **pprev_hash;
        // next_hash의 보완책인 이 필드로 인해 hash는 doubly lonked list로 동작할 수 있다.
    struct buffer_head *buffers;
        // 만약 page가 연결되어 있는 block device를 위한 buffer를 가지고 있다면
        // buffer_head에 대한 정보를 유지하기 위해 사용된다.
        // 만약 (it is backed by a swap file)이면 프로세스에 의해 mapped된 anonymous page인 경우엔
        // 연결되어 있는 buffer_head 를 가리킬 수도 있다.
        // 이때 page는, 속해있는 파일시스템에서 정하는 block size단위로,
        // 저장장치로 저장되는 sync작업이 일어나야 하므로 필요하다.
#if defined (CONFIG_HIGHMEM) || defined(WAANT_PAGE_VIRTUAL)
    void *virtual;
        // 일반적으로는 ZONE_NORMAL에 속해있는 page만이 커널에 의해 직접 지정이 가능하다
        // ZONE_HIGHMEM zone내에 있는 page를 주소지정 하기 위해 kmap()함수가 사용되며
        // 이 함수는 page를 kernel과 map시켜주는 역할을 한다.(9장에서 논의됨)
        // 고정된 개수의 page만이 map 될 수 있다. 만약 page가 map 되었다면, 이 필드는 page의 가상주소를 나타낸다.
#endif /*CONFIG_HIGHMEM || WANT_PAGE_VIRTUAL */
}mem_map_t;
```

ZONE wait queue



Buddy Allocator

- To reduce “external fragmentation”
- Fast allocation and de-allocation of pages
- 연속적인 page(block) 단위 별로 관리
  - ✓ 1, 2, 4, 8.. pages → each one block

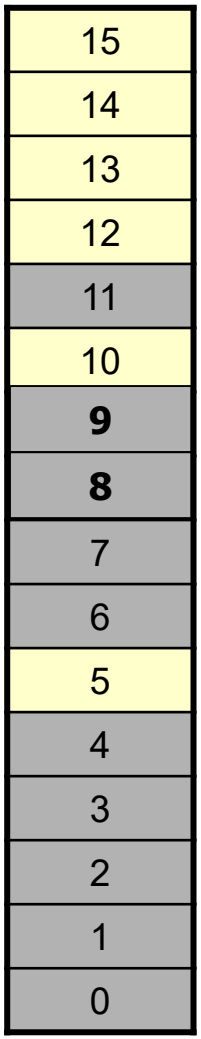
free_area_t 구조체


```
/* ~/include/linux/mmzone.h
#define    MAX_ORDER        10

typedef struct zone_struct {
        ...
        free_area_t        free_area[MAX_ORDER];
        ...
} zone_t;

typedef struct free_area_struct {
                struct list_head                free_list;
                unsigned long                   *map;
}free_area_t;
```

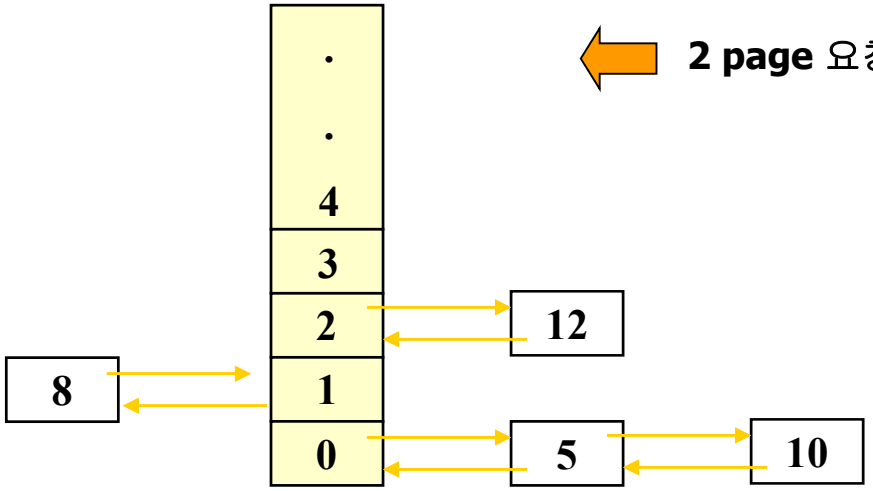
Data structure

- free_area.[order].free_list
  - ✓ 특정 크기의 free page block들의 이중 연결 리스트
- free_area.[order].map
  - ✓ ((number of pages) -1) >> (order + 4)) + 1bytes
  - ✓ 한 비트가 하나의 버디의 상태를 나타낸다

Memory allocation



Physical Memory

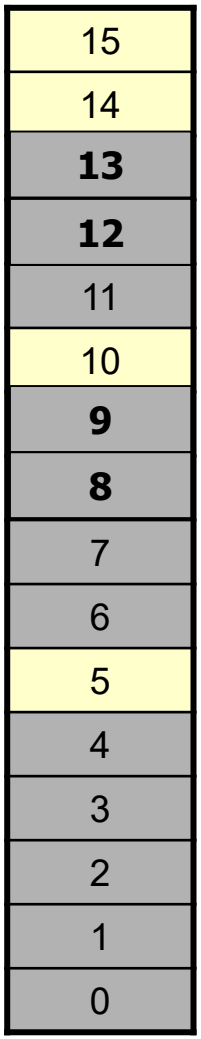


free_area.free_list

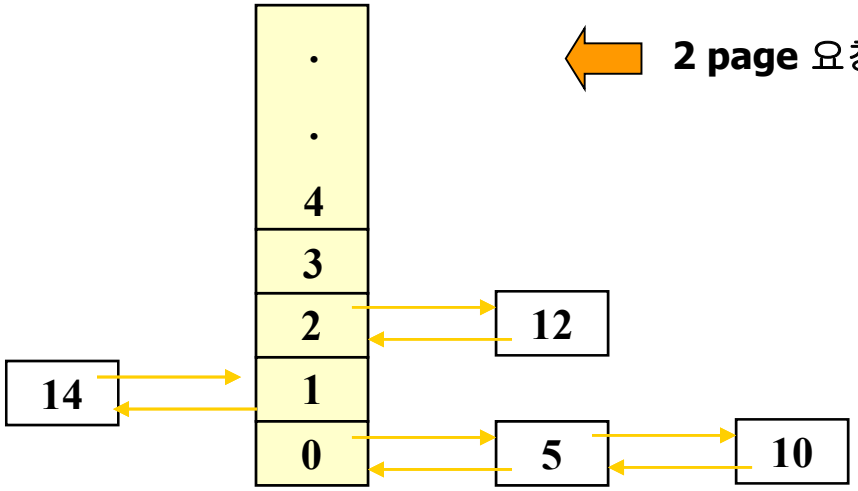
free_area[order].map

Pages	0	1	2	3	4	5	6	7	**8**	**9**	10	11	12	13	14	15
order(0)	0		0		1		0		0		1		0		0	
order(1)	0				0				1-> **0**				0			
order(2)	0								1							
order(3)	0															

Memory allocation



Physical Memory

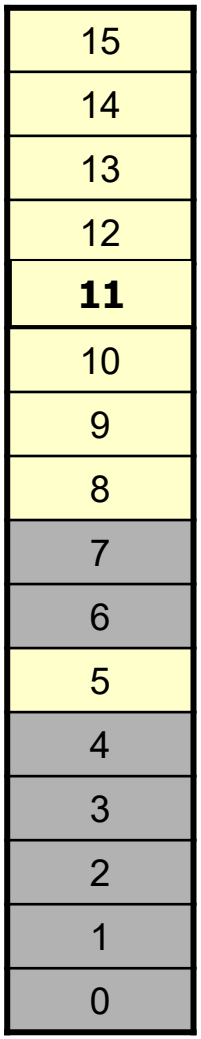


free_area.free_list

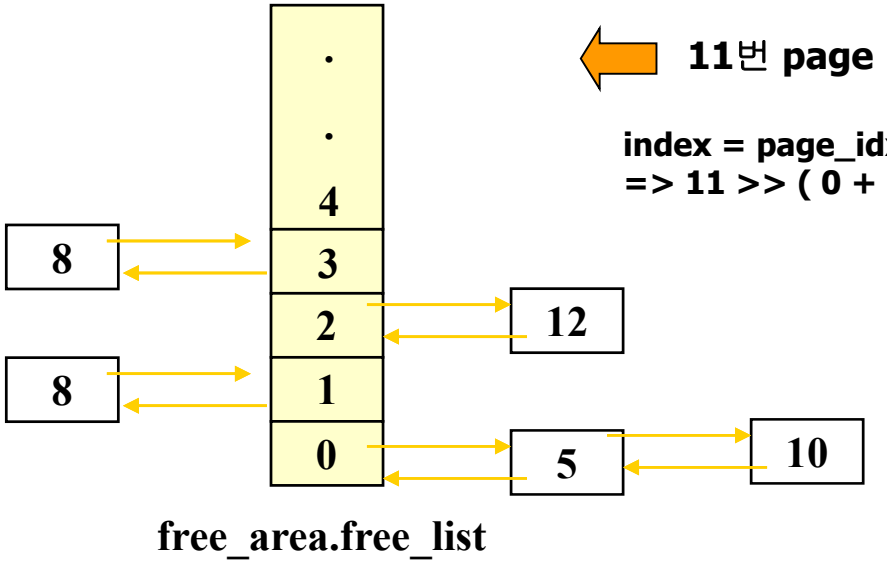
free_area[order].map

Pages	0	1	2	3	4	5	6	7	**8**	**9**	10	11	**12**	**13**	14	15
order(0)	0		0		1		0		0		1		0		0	
order(1)	0				0				0				0 **-> 1**			
order(2)	0								1 **-> 0**							
order(3)	0															

Memory de-allocation



Physical Memory



free_area[order].map

index >>= 1

Pages	0	1	2	3	4	5	6	7	8	9	10	**11**	12	13	14	15
order(0)	0		0		1		0		0		1-> **0**		0		0	
order(1)	0				0				1 -> **0**				0			
order(2)	0								1-> **0**							
order(3)	0 -> **1**															

Buddy 관련 함수

`struct page * alloc_page(unsigned int gfp_mask)`
Allocates a single page and returns a struct address.

`struct page * alloc_pages(unsigned int gfp_mask, unsigned int order)`
Allocates 2^{order} number of pages and returns a struct page.

`unsigned long get_free_page(unsigned int gfp_mask)`
Allocates a single page, zeros it, and returns a virtual address.

`unsigned long __get_free_page(unsigned int gfp_mask)`
Allocates a single page and returns a virtual address.

`unsigned long __get_free_pages(unsigned int gfp_mask, unsigned int order)`
Allocates 2^{order} number of pages and returns a virtual address.

`struct page * __get_dma_pages(unsigned int gfp_mask, unsigned int order)`
Allocates 2^{order} number of pages from the DMA zone and returns a struct page.

Table 6.1. Physical Pages Allocation API

`void __free_pages(struct page *page, unsigned int order)`
Frees an order number of pages from the given page.

`void __free_page(struct page *page)`
Frees a single page.

`void free_page(void *addr)`
Frees a page from the given virtual address.

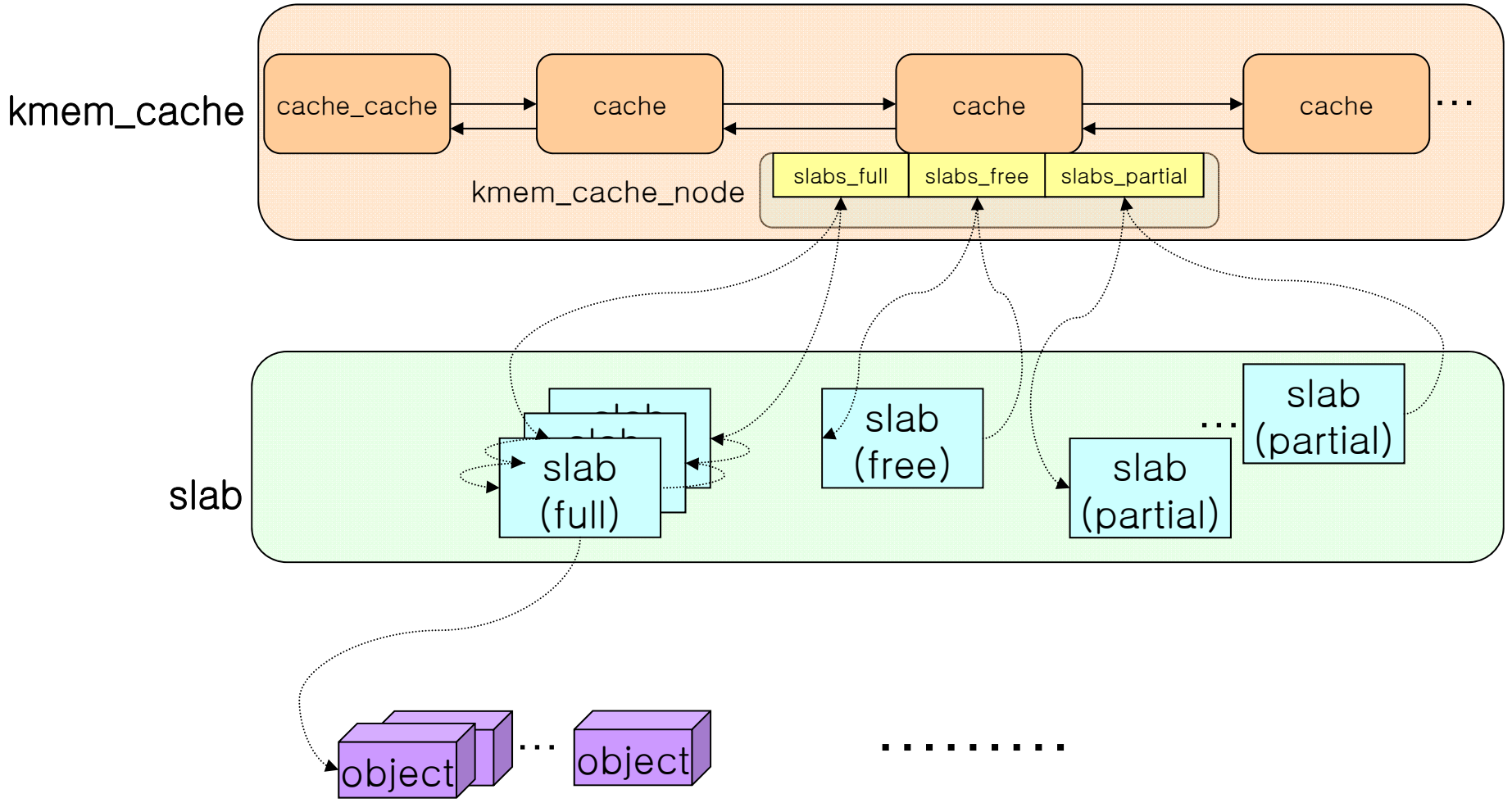
Table 6.2. Physical Pages Free API

Slab Allocator

- 슬랩 할당자
  - ✓ 버디알고리즘은 적은 메모리 할당엔 부적절
  - ✓ 같은 크기의 메모리 공간에 대한 할당을 반복하는 경향 있으므로 캐시 개념 도입

  - ✓ 캐시 : 객체의 모음 ➔ 같은 크기 메모리 공간의 창고
  - ✓ 슬랩 : 캐시가 들어있는 주 메모리 영역을 나눈것**(/proc/slabinfo)**
    - 연속된 PF하나이상으로 구성됨
    - 할당한 객체와 여유객체를 모두 포함
    - 빈 슬랩의 PF스스로 해제 안함

Slab Allocator 구조



슬랩 관련 함수

`kmem_cache_t * kmem_cache_create(const char *name, size_t size, size_t offset, unsigned long flags, void (*ctor)(void*, kmem_cache_t *, unsigned long), void (*dtor)(void*, kmem_cache_t *, unsigned long))`
Creates a new cache and adds it to the cache chain.

`int kmem_cache_reap(int gfp_mask)`
Scans at most `REAP_SCANLEN` caches and selects one for reaping all per-cpu objects and free slabs from. It is called when memory is tight.

`int kmem_cache_shrink(kmem_cache_t *cachep)`
This function will delete all per-cpu objects associated with a cache and delete all slabs in the `slabs_free` list. It returns the number of pages freed.

`void * kmem_cache_alloc(kmem_cache_t *cachep, int flags)`
Allocates a single object from the cache and returns it to the caller.

`void kmem_cache_free(kmem_cache_t *cachep, void *objp)`
Frees an object and returns it to the cache.

`void * kmalloc(size_t size, int flags)`
Allocates a block of memory from one of the sizes cache.

`void kfree(const void *objp)`
Frees a block of memory allocated with kmalloc.

`int kmem_cache_destroy(kmem_cache_t * cachep)`
Destroys all objects in all slabs and frees up all associated memory before removing the cache from the chain.

Table 8.1. Slab Allocator API for Caches

GFP_MASK

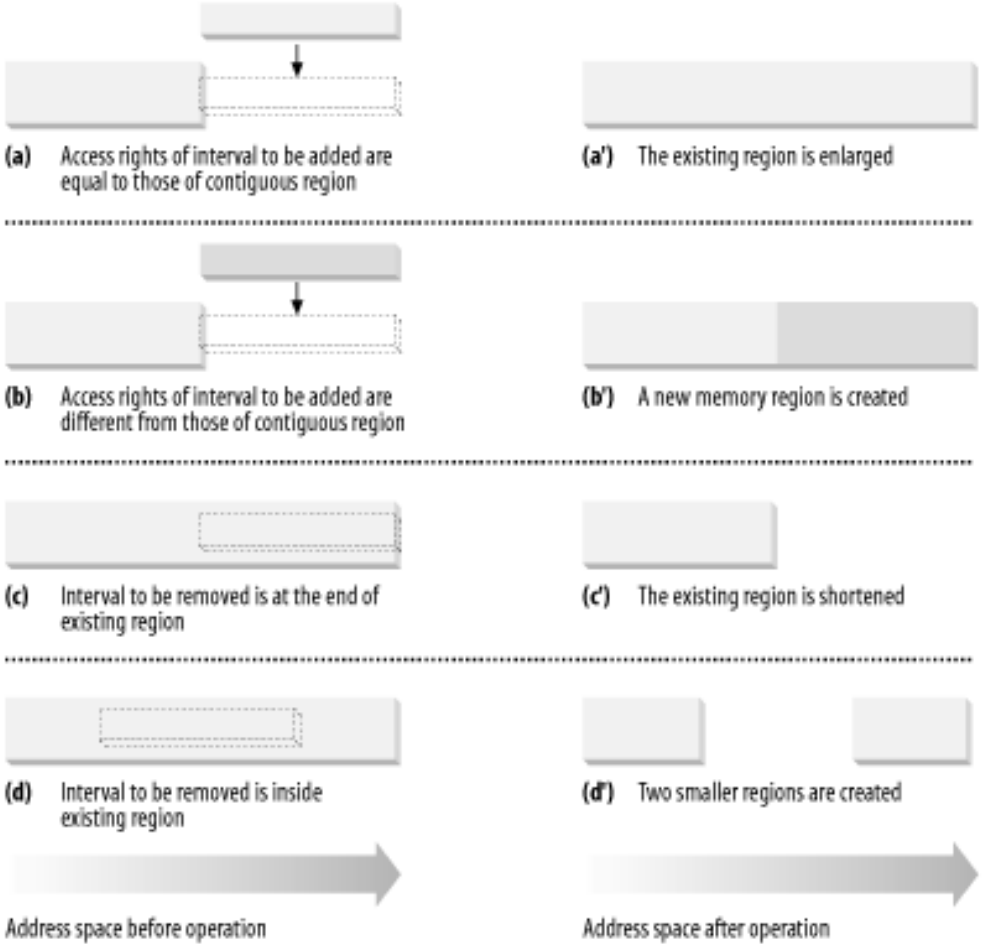
Group name	Corresponding flags
GFP_ATOMIC	__GFP_HIGH
GFP_NOIO	__GFP_HIGH __GFP_WAIT
GFP_NOHIGHIO	__GFP_HIGH __GFP_WAIT __GFP_IO
GFP_NOFS	__GFP_HIGH __GFP_WAIT __GFP_IO __GFP_HIGHIO
GFP_KERNEL	__GFP_HIGH __GFP_WAIT __GFP_IO __GFP_HIGHIO __GFP_FS
GFP_NFS	__GFP_HIGH __GFP_WAIT __GFP_IO __GFP_HIGHIO __GFP_FS
GFP_KSWAPD	__GFP_WAIT __GFP_IO __GFP_HIGHIO __GFP_FS
GFP_USER	__GFP_WAIT __GFP_IO __GFP_HIGHIO __GFP_FS
GFP_HIGHUSER	__GFP_WAIT __GFP_IO __GFP_HIGHIO __GFP_FS __GFP_HIGHMEM

Table 7-6. Zone modifier lists

__GFP_DMA	__GFP_HIGHMEM	Zone list
0	0	ZONE_NORMAL + ZONE_DMA
0	1	ZONE_HIGHMEM + ZONE_NORMAL + ZONE_DMA
1	0	ZONE_DMA
1	1	ZONE_DMA

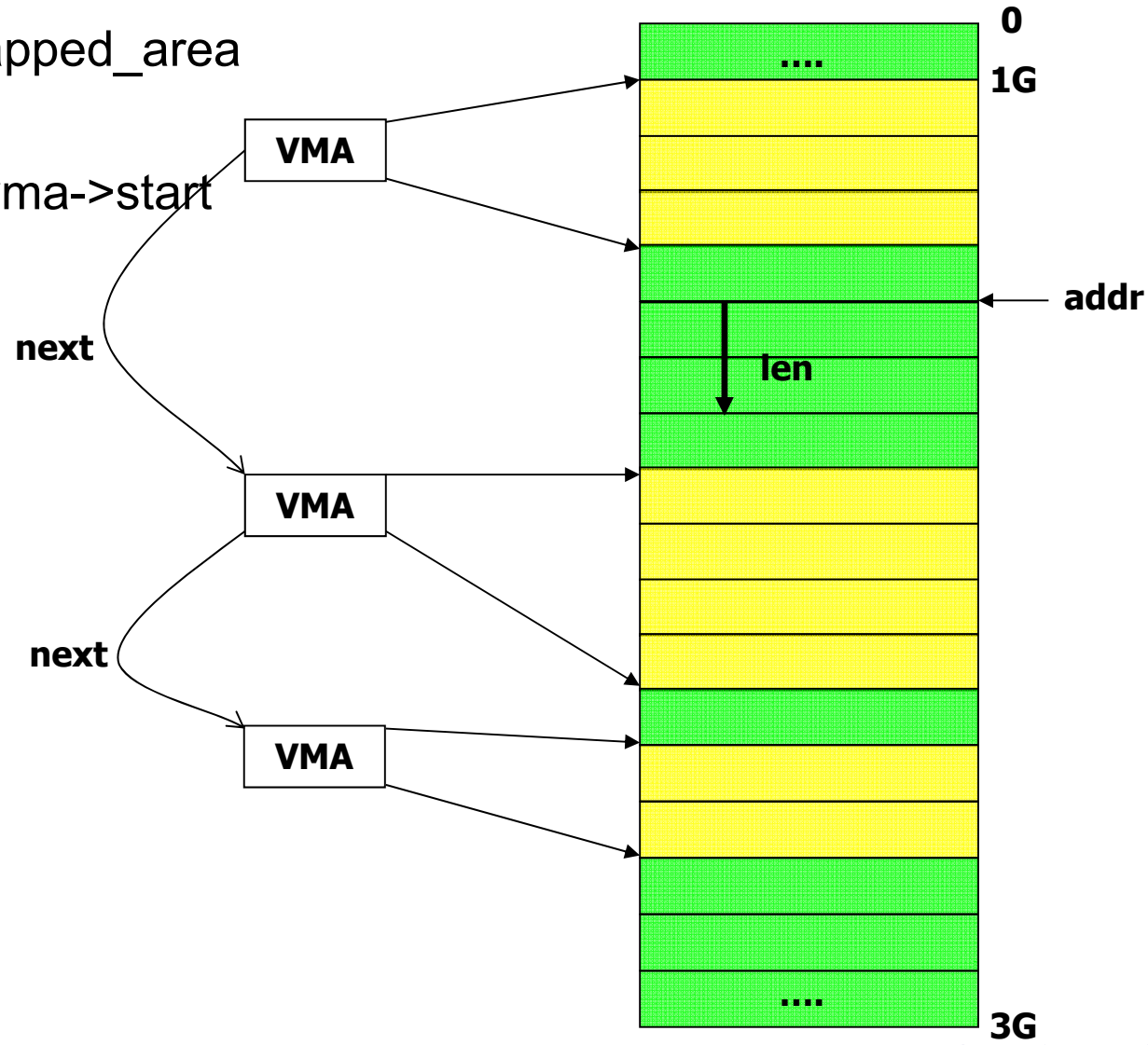
가상 주소 공간 할당/해제

- 일부 가상 주소 공간 = region = 영역 = 구간 = vm_area_struct
- 메모리 구역 겹치는 일은 없음
- 인접한 두 구역은 접근 권한 일치 시 합침



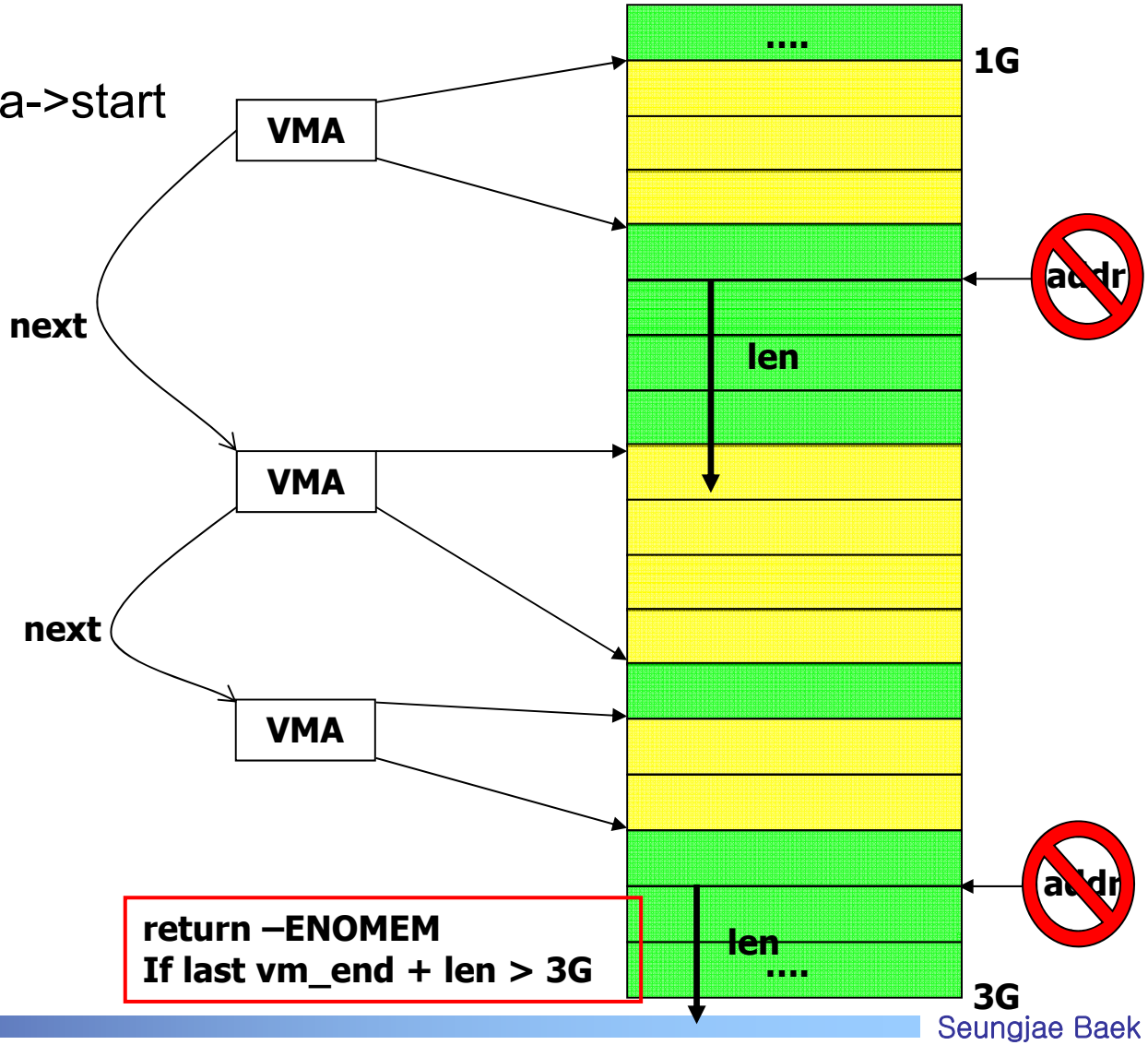
Creating a memory region ex.

- case 1
  - ✓ arch_get_unmapped_area
  - ✓ if addr != null
  - ✓ if addr + len < vma->start



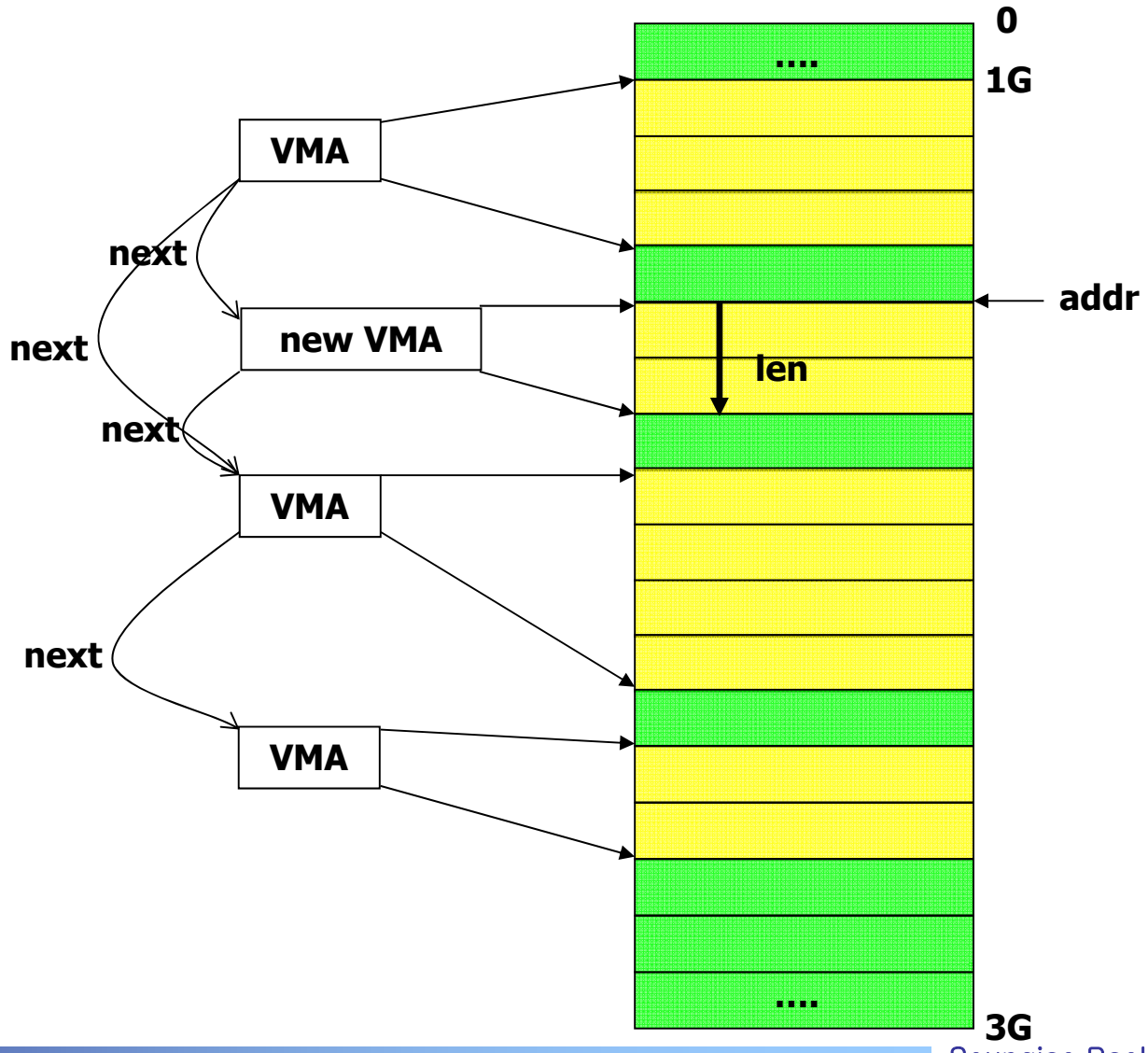
Creating a memory region ex.

- case 2
  - ✓ addr != null
  - ✓ addr + len > vma->start
  - ✓ OR addr == null



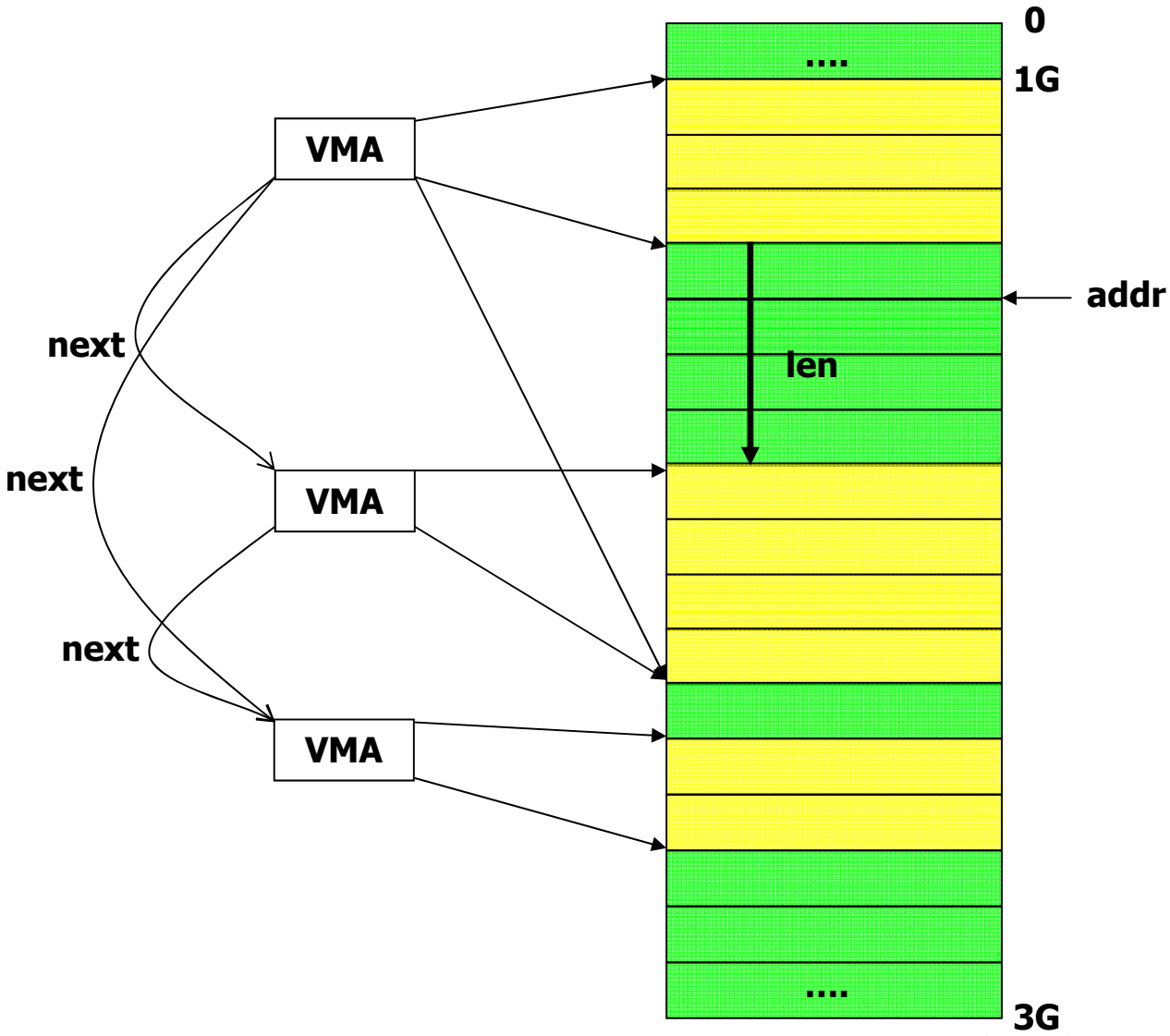
Creating a memory region ex.

- do_mmap_pgoff



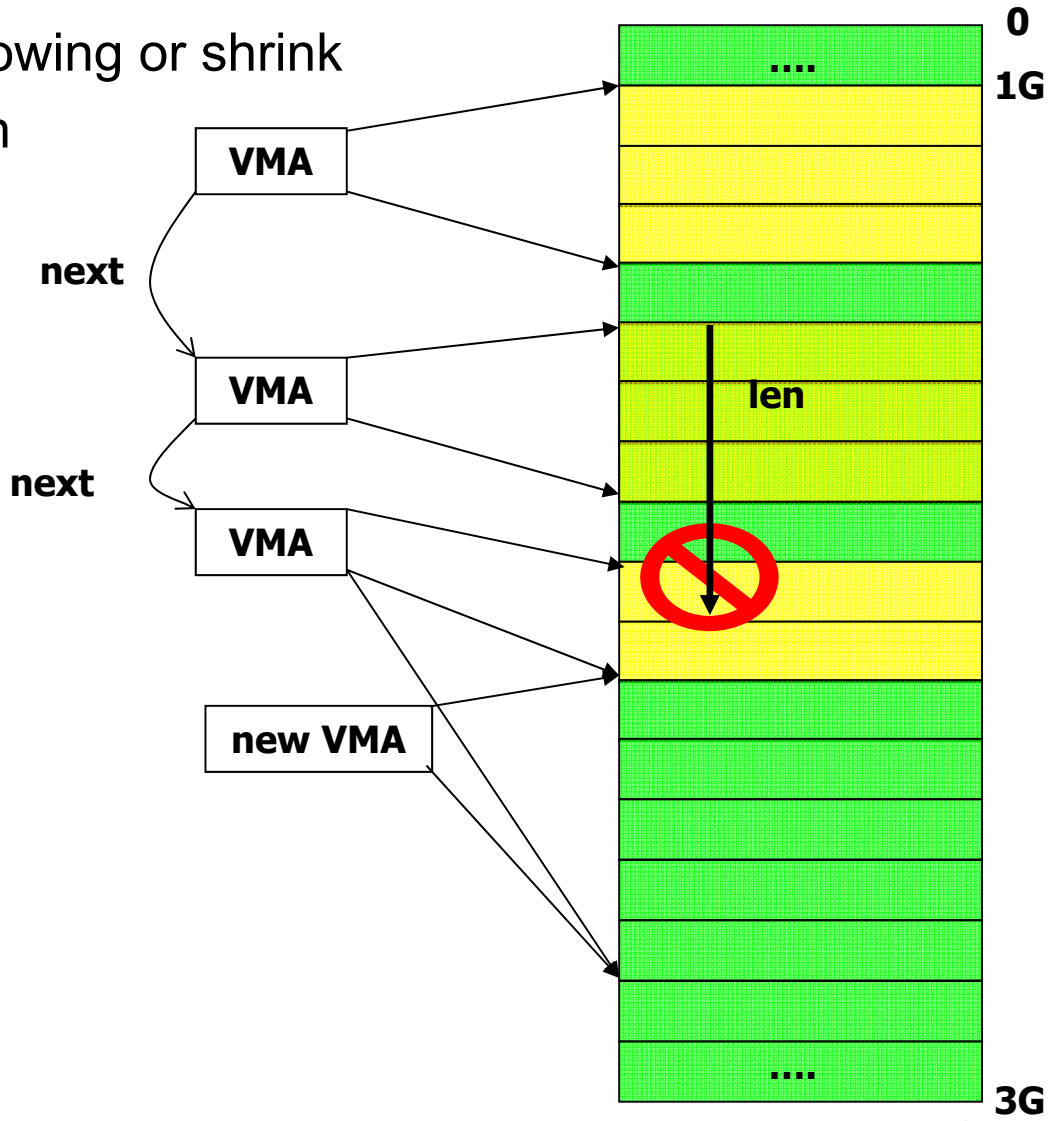
Merging contiguous region

- do_mmap_pgoff
  - ✓ merge-able



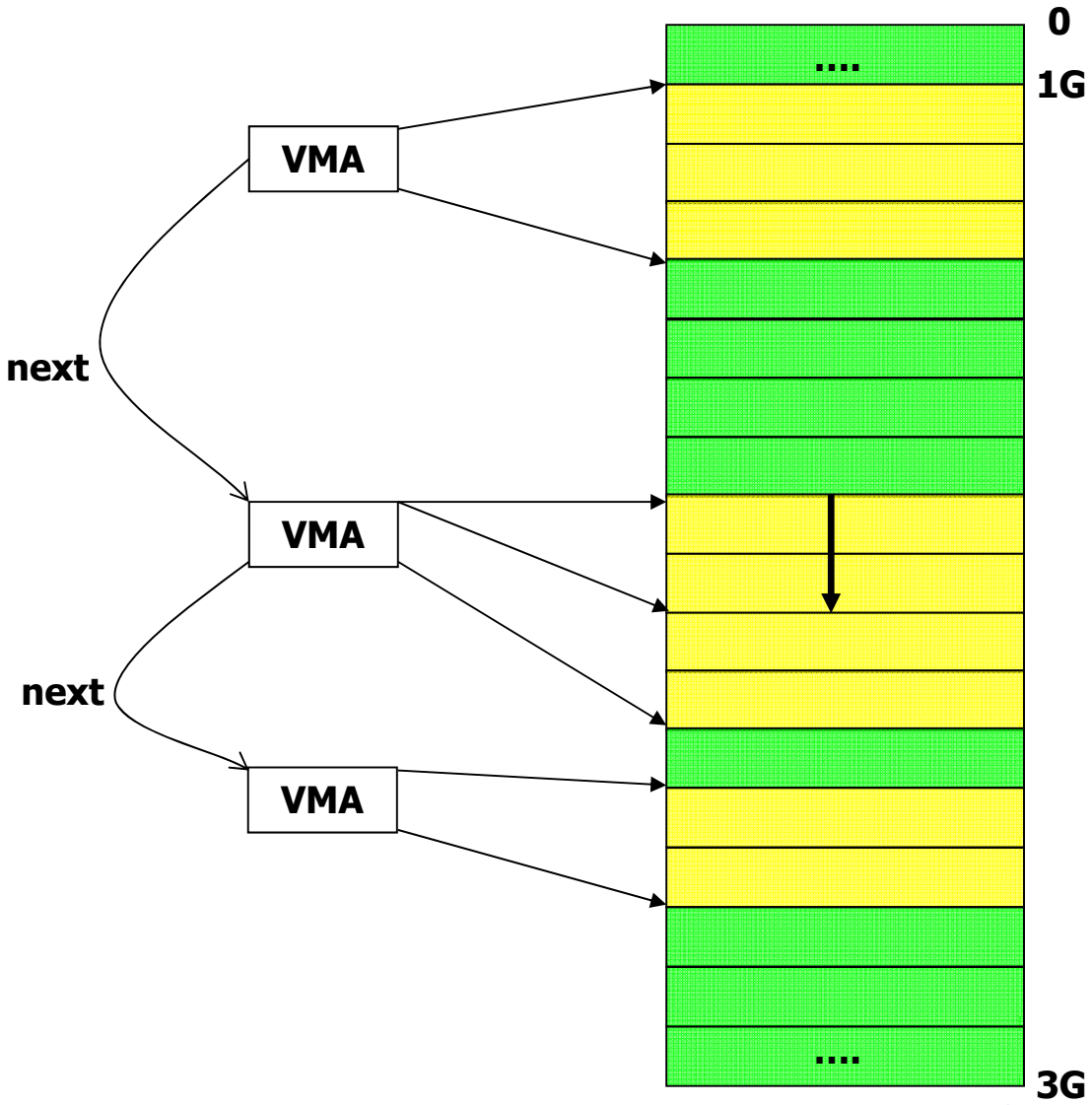
Remapping and Moving a memory region

- do_mmap_pgoff
  - ✓ memory region growing or shrink
  - ✓ find another region



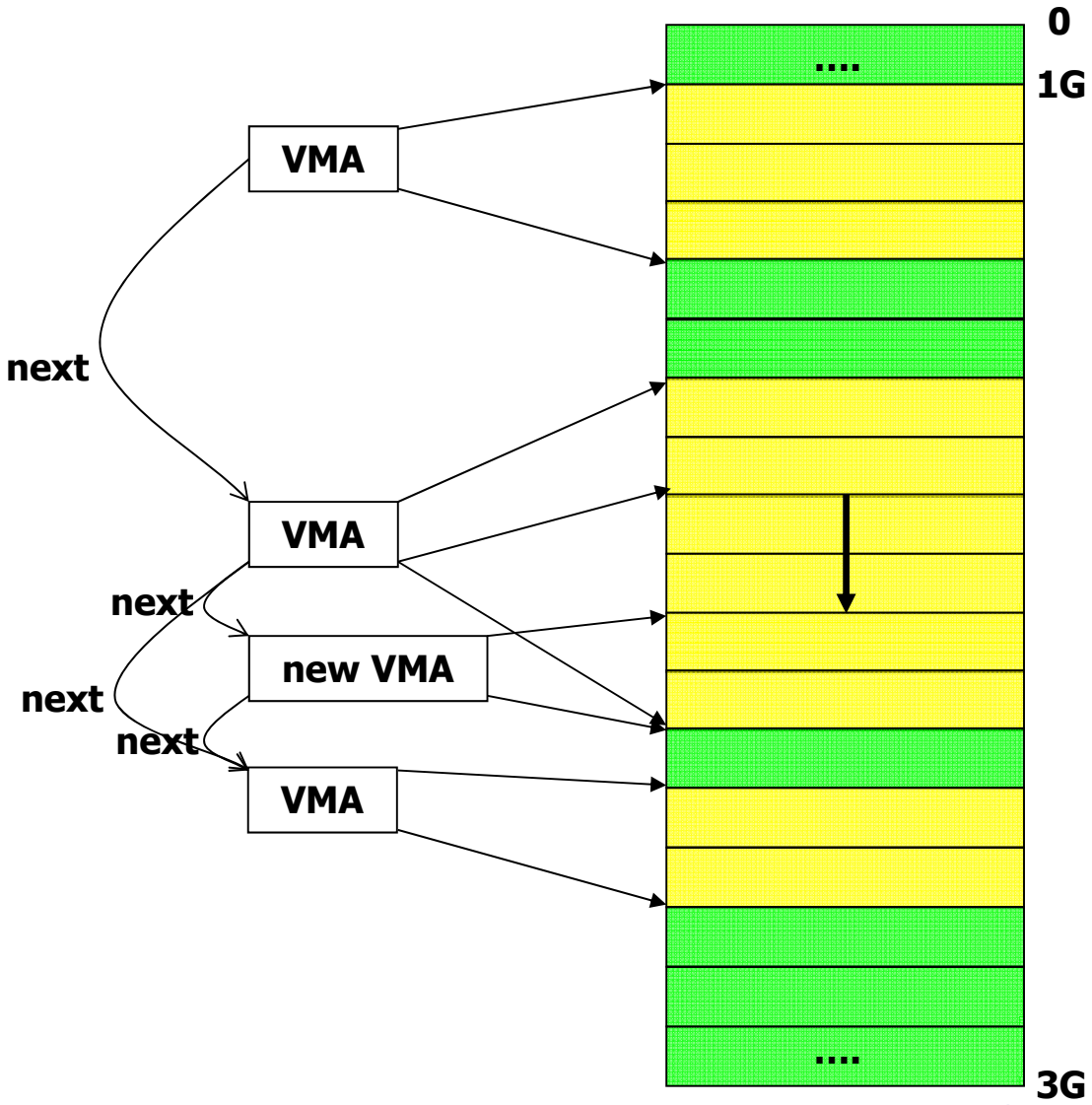
Delete a memory region

- do_munmap



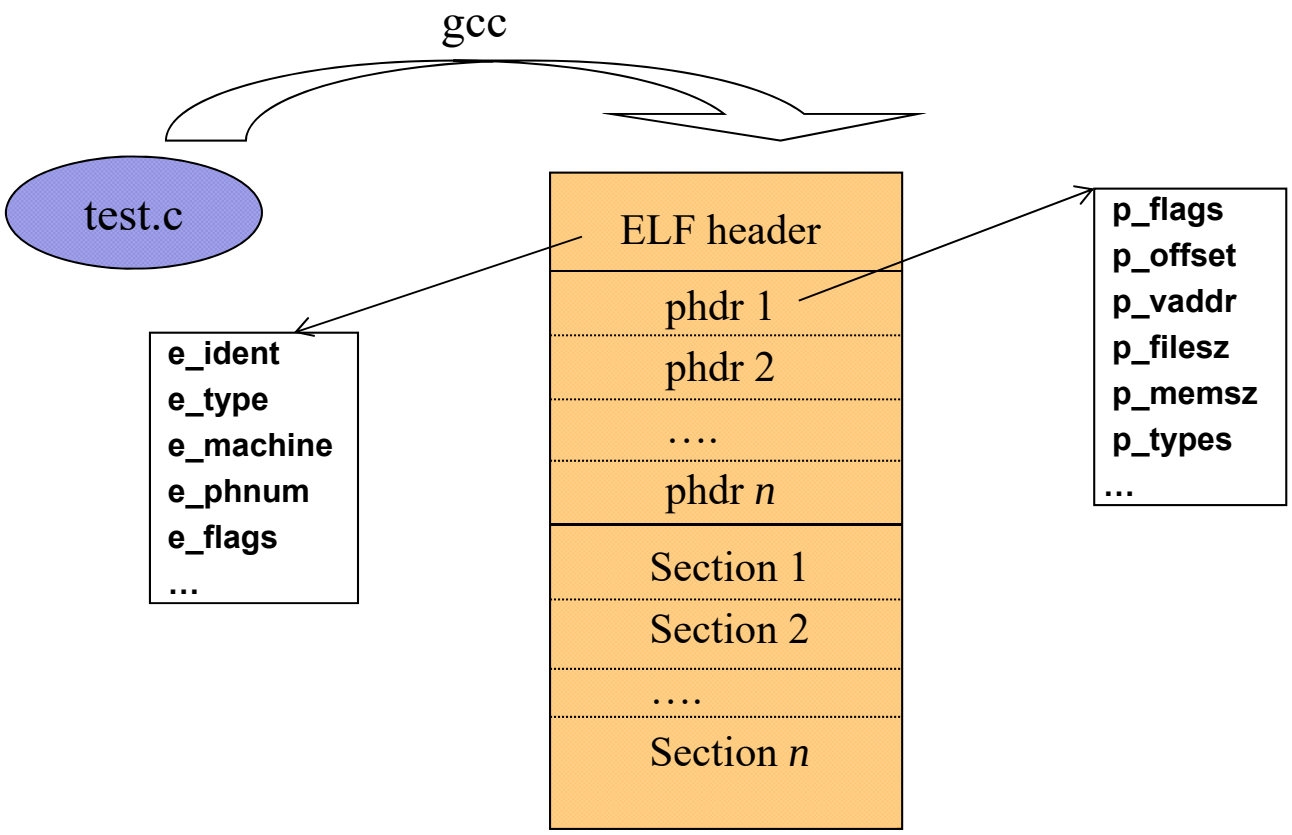
Delete a memory region

- do_munmap



간단한 ELF 구조

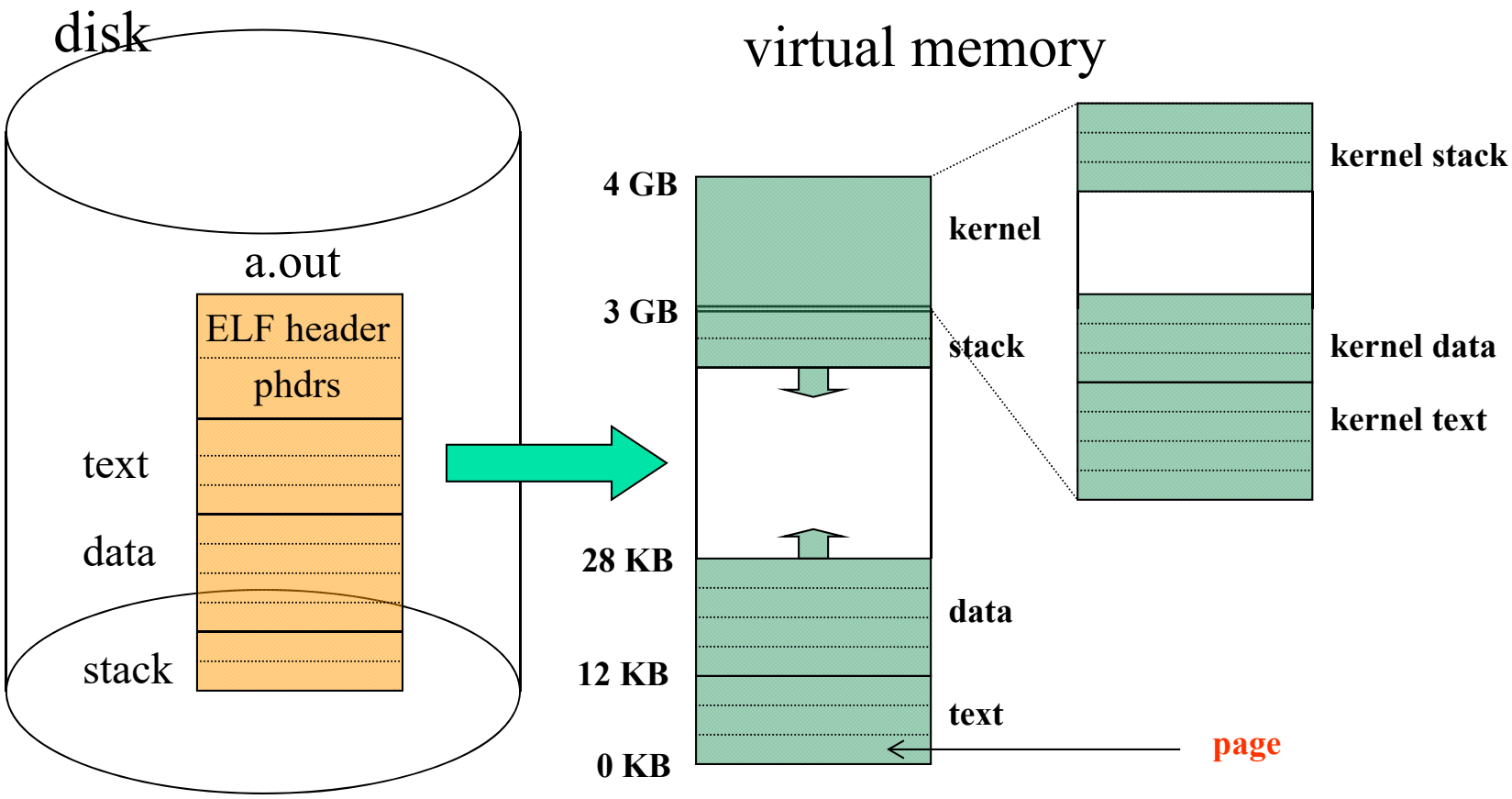
- test.c의 컴파일 결과 : ELF format



a.out : (ELF format)
/*include/linux/elf.h */

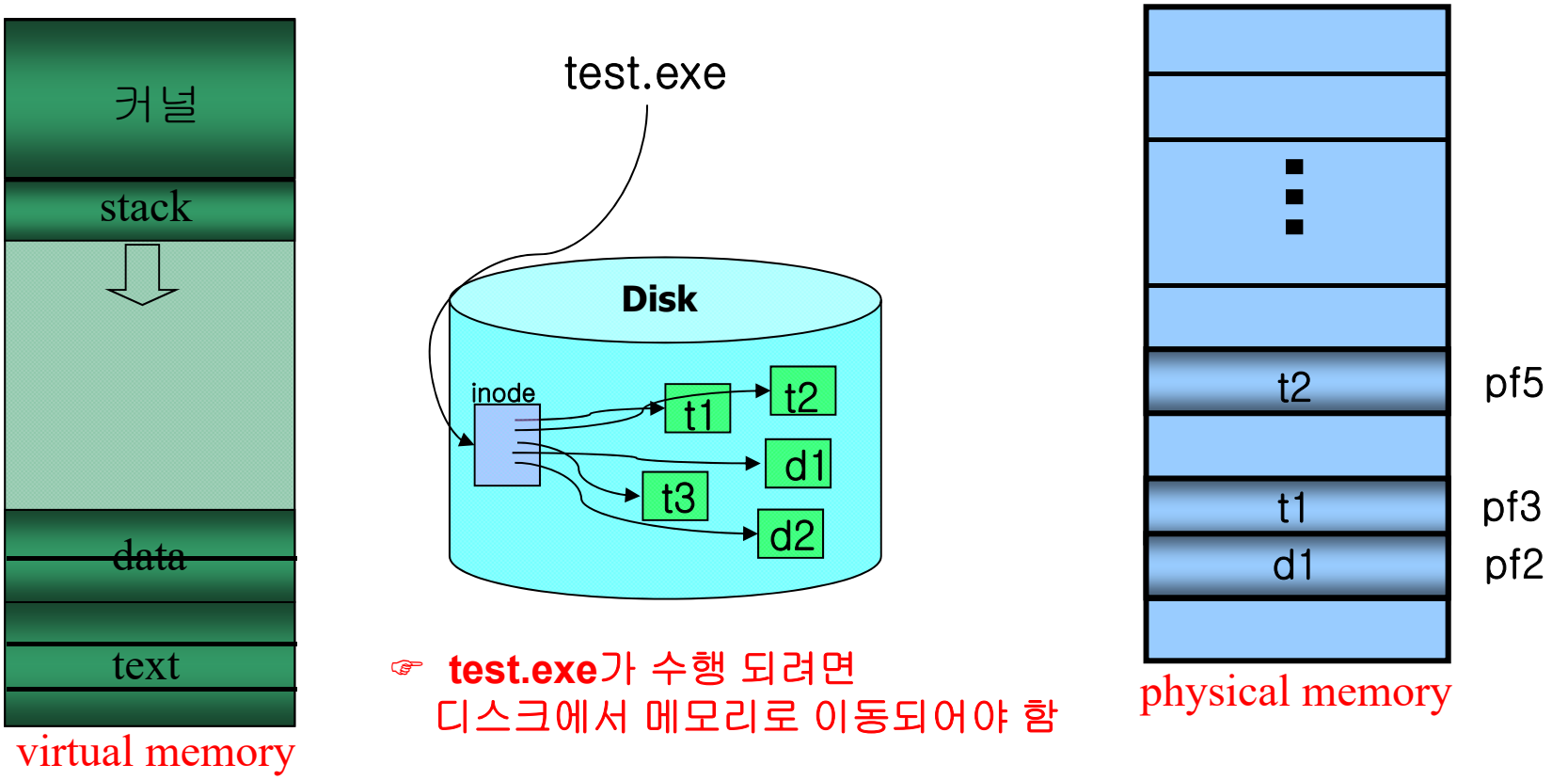
가상 메모리와 ELF 파일

- 가상 메모리와 ELF format의 대응



가상 메모리와 태스크 실행

- 페이지 시스템 : 태스크 로딩 (execve())
  - ✓ test.exe라는 프로그램을 수행 시킨다고 가정 (ELF format)
    - 이 응용의 text는 12KB, data는 8KB, 초기 stack은 4KB로 가정
    - 이 응용은 malloc을 사용하지 않음

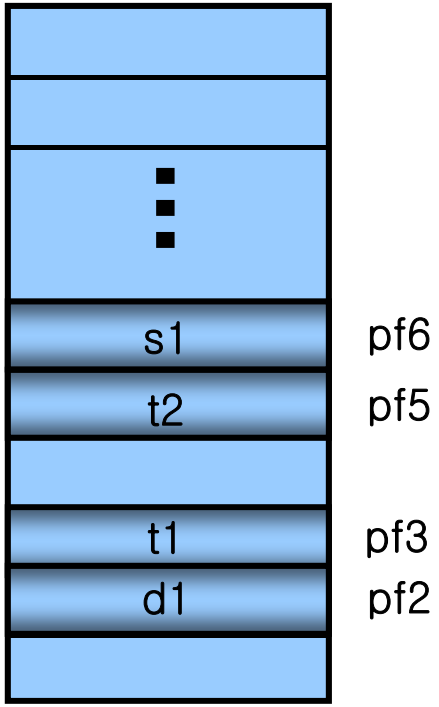


태스크와 페이지 테이블

- 페이지 시스템 : 주소 변환
  - ✓ 디스크 내용을 물리 메모리에 로딩(loading) 하면서 위치 정보 기록

page	page frame
t1	**pf3**
t2	**pf5**
t3	**-**
d1	**pf2**
d2	**-**
s1	**pf6**

Page table



physical memory

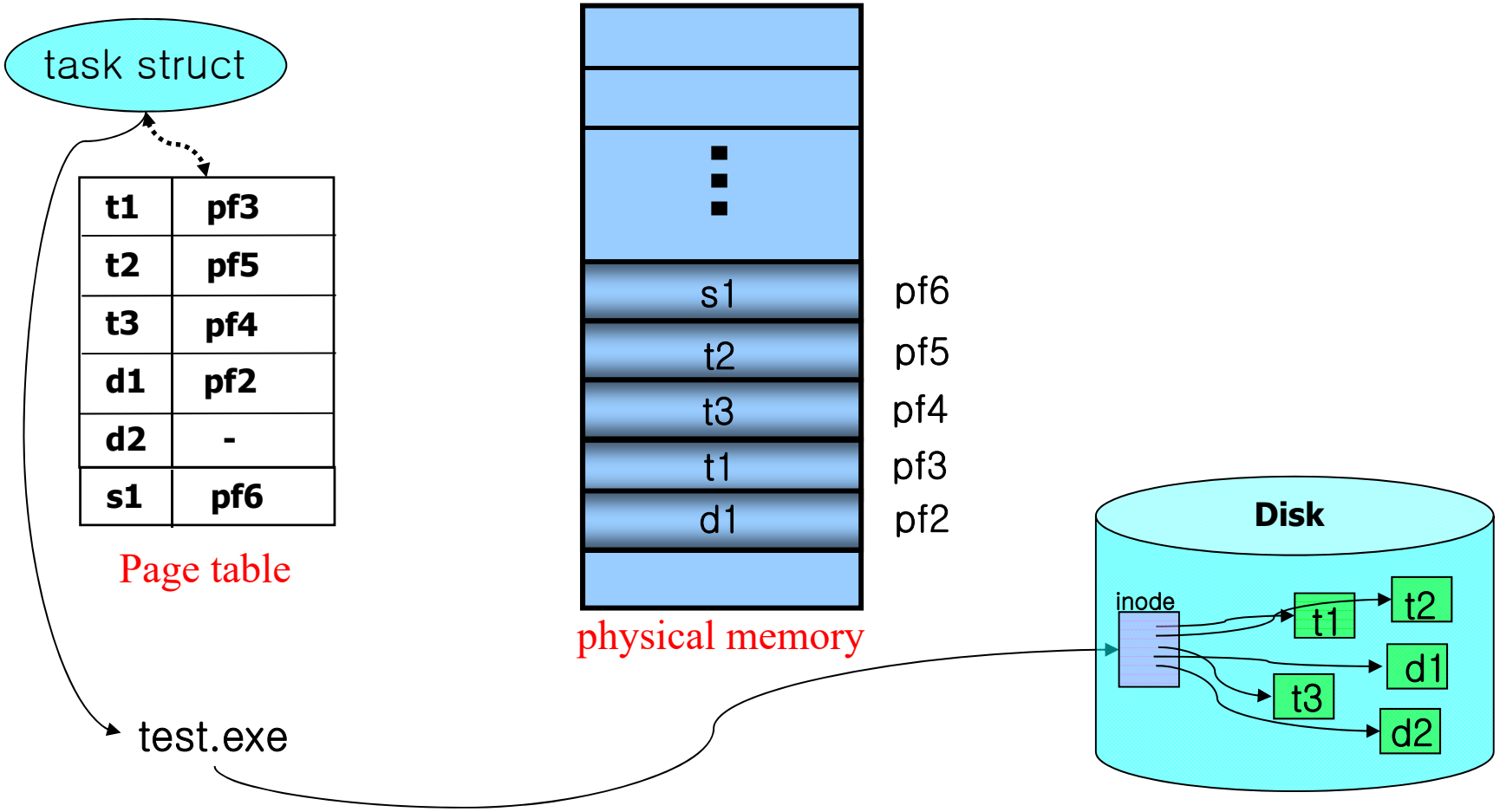
☞ 태스크 자료 구조 **(task struct)**에서 **page table**을 관리: 메모리 문맥

☞ 위의 구조에서 프로그램이 가상 주소 **1000**번지에 접근하면 메모리 어디에 접근하는가**?**

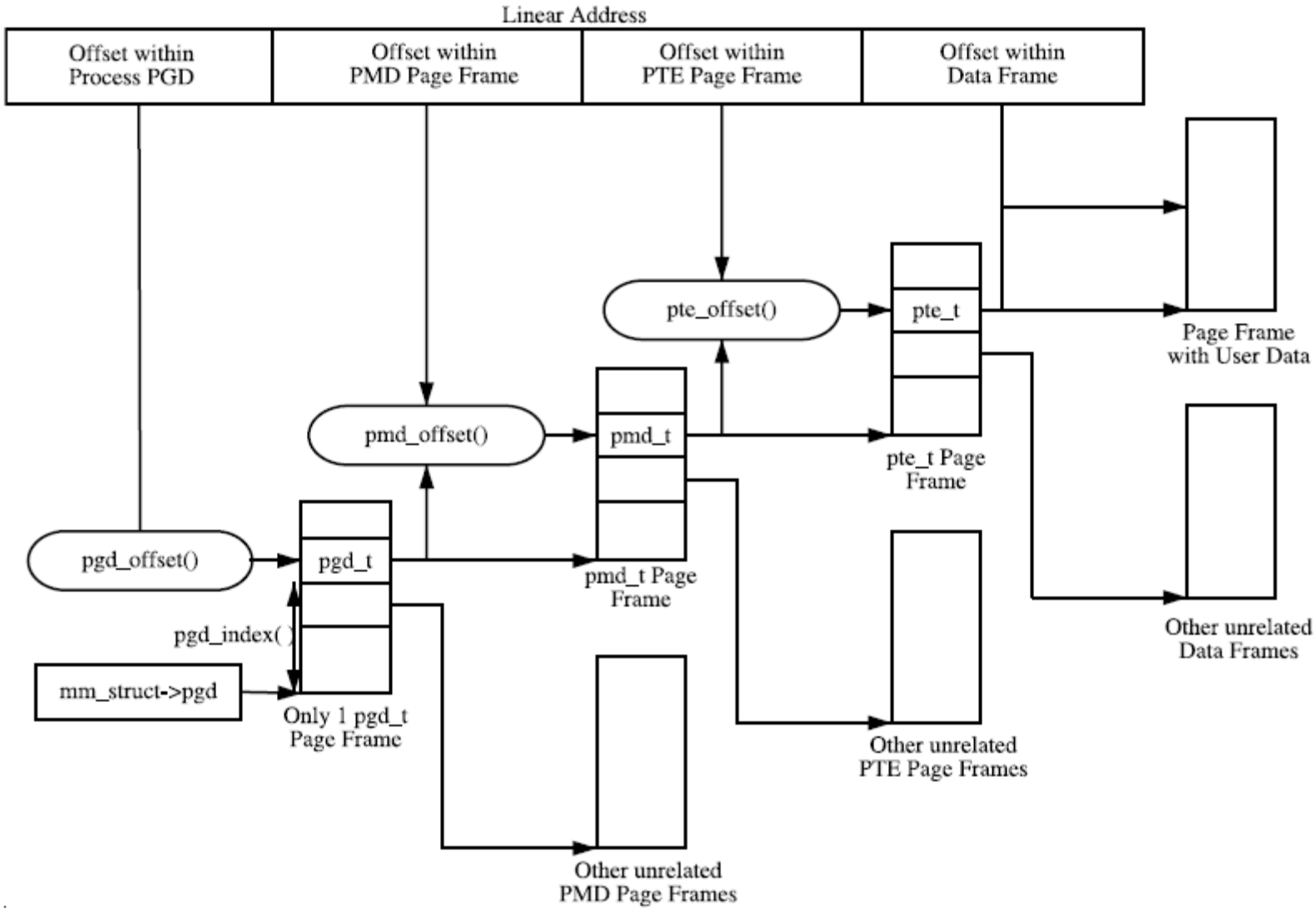
☞ 가상 주소 **5000**번지에 접근하면**?** 또한 가상 주소 **9000**에 접근하면**?**

페이지 폴트

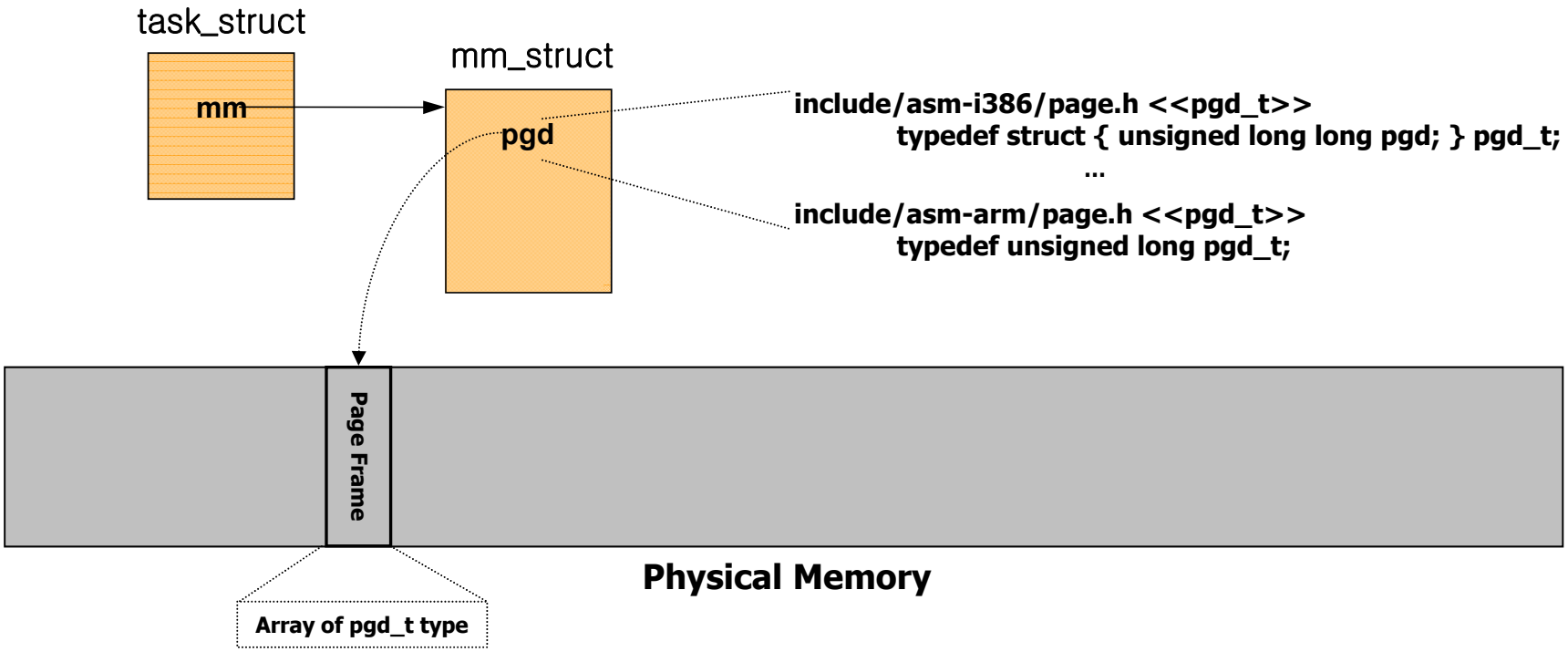
- 페이지 시스템 : 페이지 부재 결함
  - ✓ 페이지 프레임에 존재하지 않는 페이지를 접근할 때



Linux의 Page Table Management 전체 구조



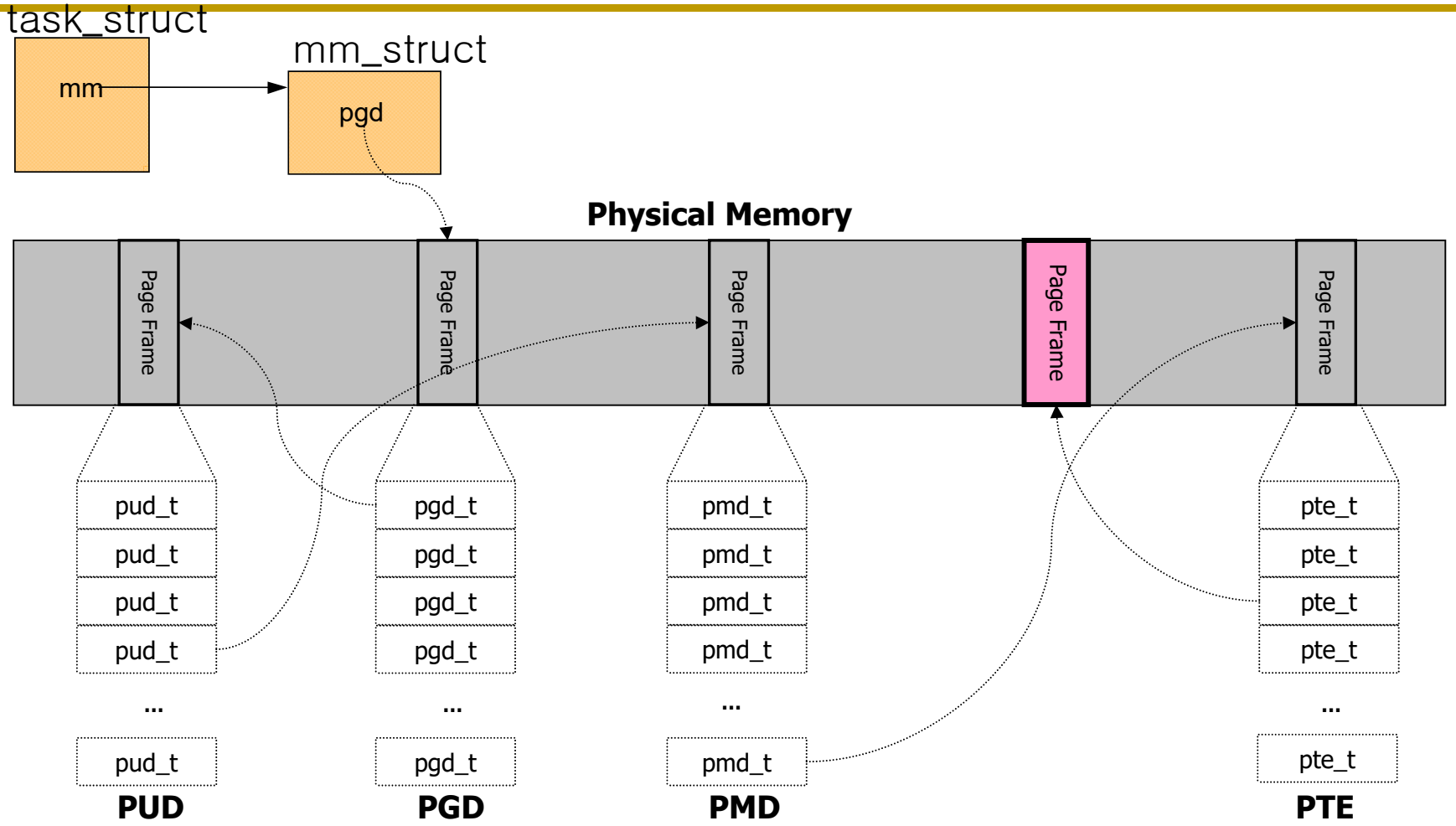
Linux의 PGD



☞**pgd 로딩?**

☞**X86** 이라면 **mm->pgd**를 **cr3**에 복사함으로써 이뤄짐 ➔ **__flush_tlb()**
☞**(cr3** 복사는 **TLB flushing**을 유발하는 단점이 있음)

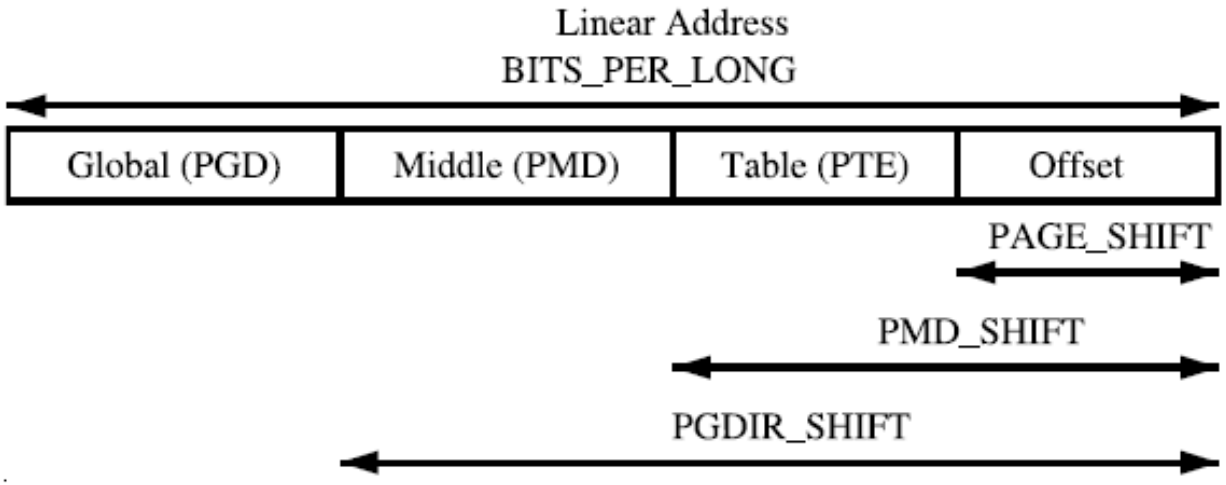
Linux의 PGD, PMD, PTE

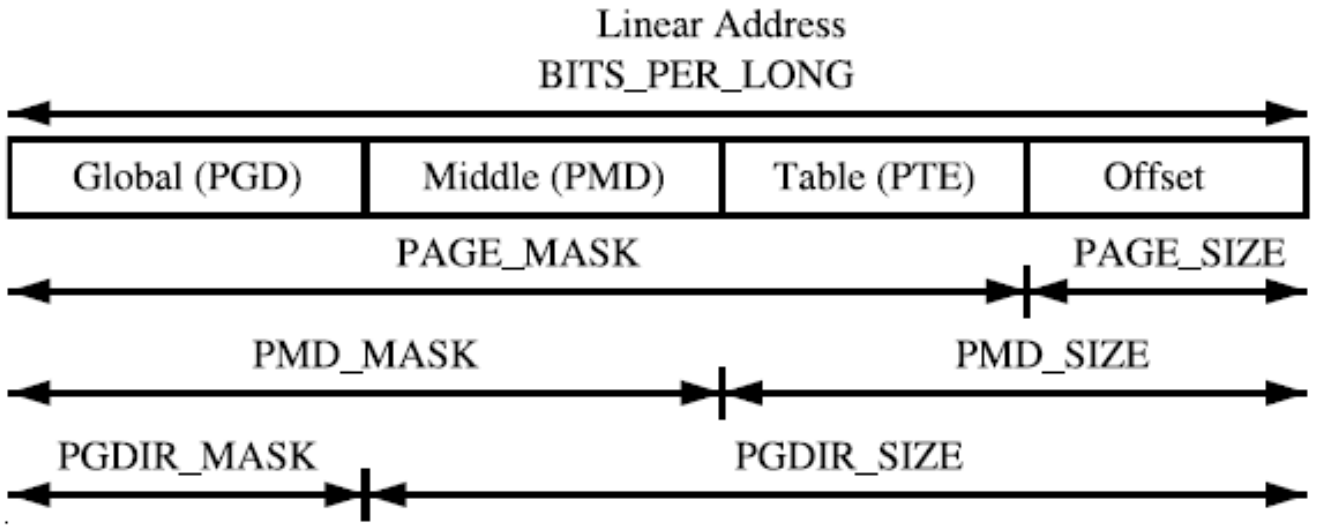


☞**Swap out**된 경우에는?

☞**Swap entry**가 **PTE**에 저장되어 있음
☞**Fault** 맞으면 이 **swap entry**를 이용해 **do_swap_page()** 함수가 **page**를 처리해 줌

주소 변환을 위한 매크로


Linear Address
BITS_PER_LONG
Global (PGD)
Middle (PMD)
Table (PTE)
Offset
PAGE_SHIFT
PMD_SHIFT
PGDIR_SHIFT

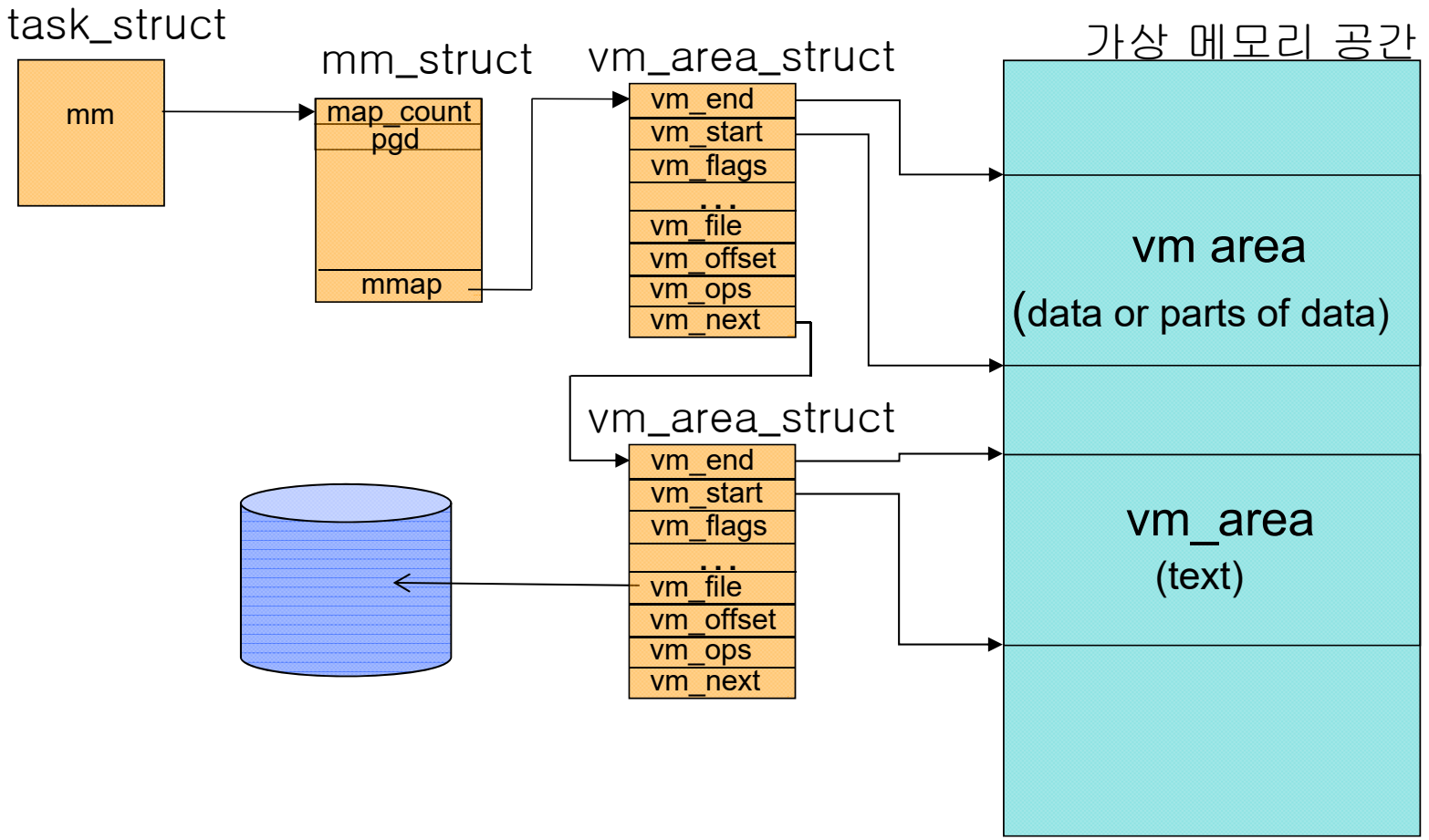

Linear Address
BITS_PER_LONG
Global (PGD)
Middle (PMD)
Table (PTE)
Offset
PAGE_MASK
PAGE_SIZE
PMD_MASK
PMD_SIZE
PGDIR_MASK
PGDIR_SIZE

주소 변환 과정의 이해


```
/* mm/memory.c */
static struct page * follow_page(struct vm_area_struct *vma, unsigned long address,
                                        unsigned int flags, unsigned int *page_mask)
{
        pgd_t *pgd;
         pud_t *pud;
        pmd_t *pmd;
        pte_t *ptep, pte;

        pgd = pgd_offset(mm, address);
        if( pgd_none(*pgd) || pgd_bad(*pgd))       goto out;
        pud = pud_offset(pgd, address);
        if( pud_nond(*pud) )                       goto out;
        pmd = pmd_offset(pgd, address);
        if( pmd_none(*pmd) || pmd_bad(*pmd))    goto out;

        …

        return follow_page_pte(vma, address, pmd, flags);
}
```

태스크와 가상 주소공간

- 태스크의 메모리 관리 전체 구조 (global view)

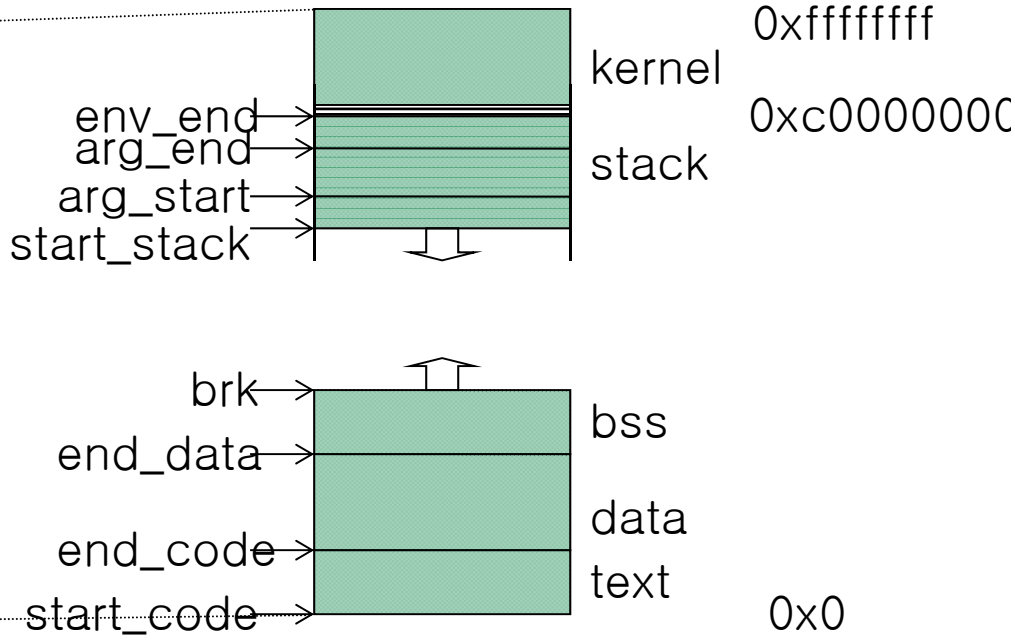


mm_struct와 가상 주소 공간 구조

- mm_struct 자료 구조 분석

include/linux/mm_types.h

```
struct mm_struct  {
    struct vm_area_struct *mmap;
    struct rb_root mm_rb;
    struct vm_area_struct *mmap_cache;
    atomic_t mm_users, mm_count
    pgd_t *pgd;
    int map_count;
    struct semaphore mmap_sem;
    mm_context_t context;
    unsigned long start_code, end_code, start_data;
    unsigned long end_data, start_brk, brk, start_stack;
    unsigned long arg_start, arg_end, env_start, env_end;
    unsigned long total_vm, locked_vm, def_flags;
    ...
}
```



include/x86/include/asm/pgtable_types.h

```
typedef struct {pgdval_t pgd;} pgd_t;
```

include/x86/asm/pgtable_64_types.h

```
typedef struct {pgdval_t pgd;} pgd_t;
```

vm_area_struct 자료 구조

❑ vm_area_struct 자료 구조 분석

include/linux/mm.h

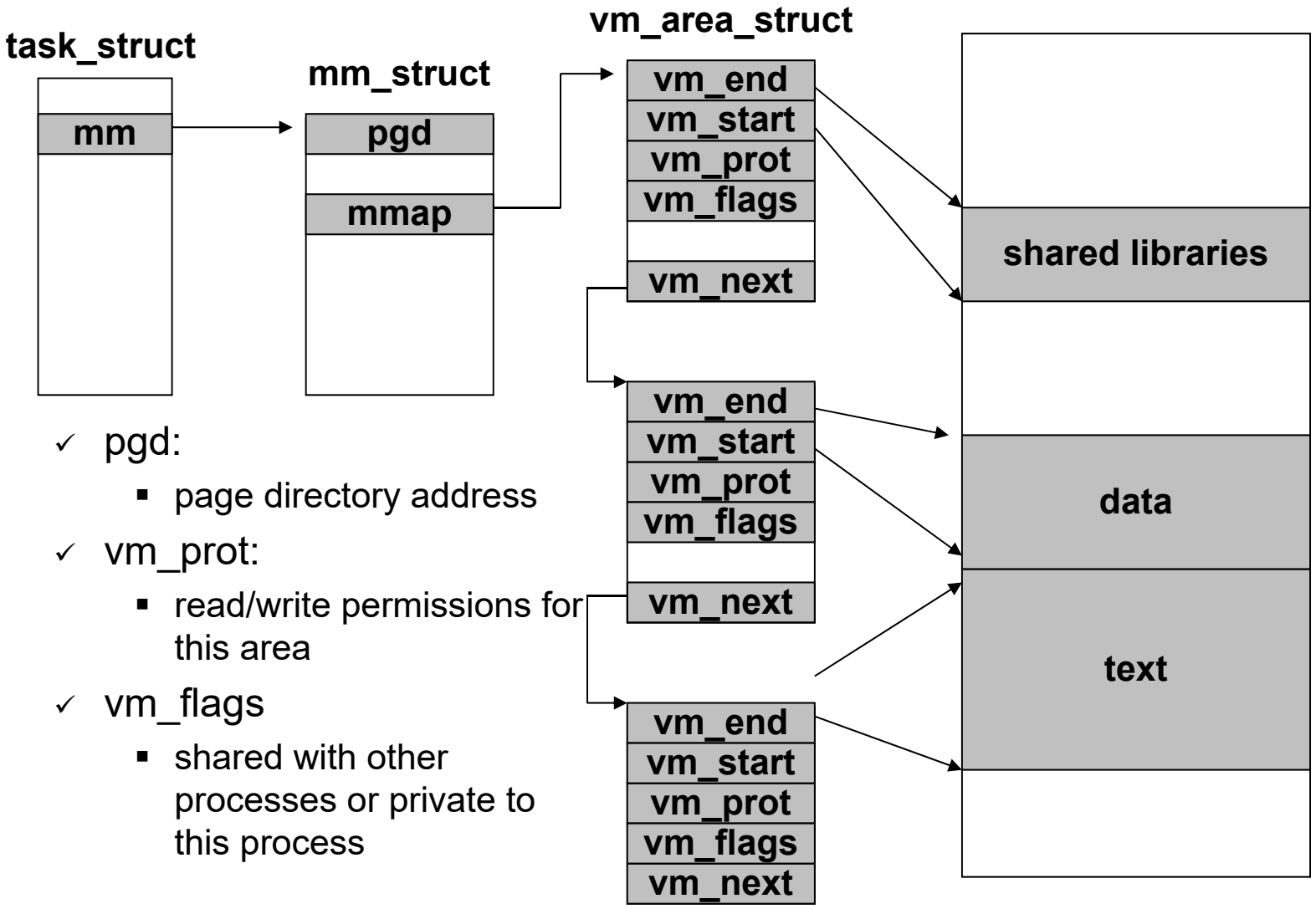
```
struct  vm_area_struct  {
    struct mm_struct *vm_mm;
    unsigned long vm_start, vm_end;
    struct vm_area_struct *vm_next
    pgprot_t vm_page_prot;
    unsigned short vm_flags;
    short  vm_avl_height;
    struct vm_area_struct *vm_avl_left;
    struct vm_area_struct *vm_avl_right;
    struct vm_area_struct *vm_next_share;

    struct vm_operations_struct  *vm_ops;
    unsigned long vm_offset;
    struct file *vm_file;
    unsigned long vm_pte;  /* for SVR4 SM */
}
```

Virtual Memory Area

PAGE_SHARED (COPY, READONLY, KERNEL)

- **open(vm_area)**
- **close(vm_area)**
- **do_mmap(file, addr, len, prot, flags, off)**
- **unmap()**
- **protect()**
- **nopage()**
- **wppage()**
- **swapout()**
- **swapin()**

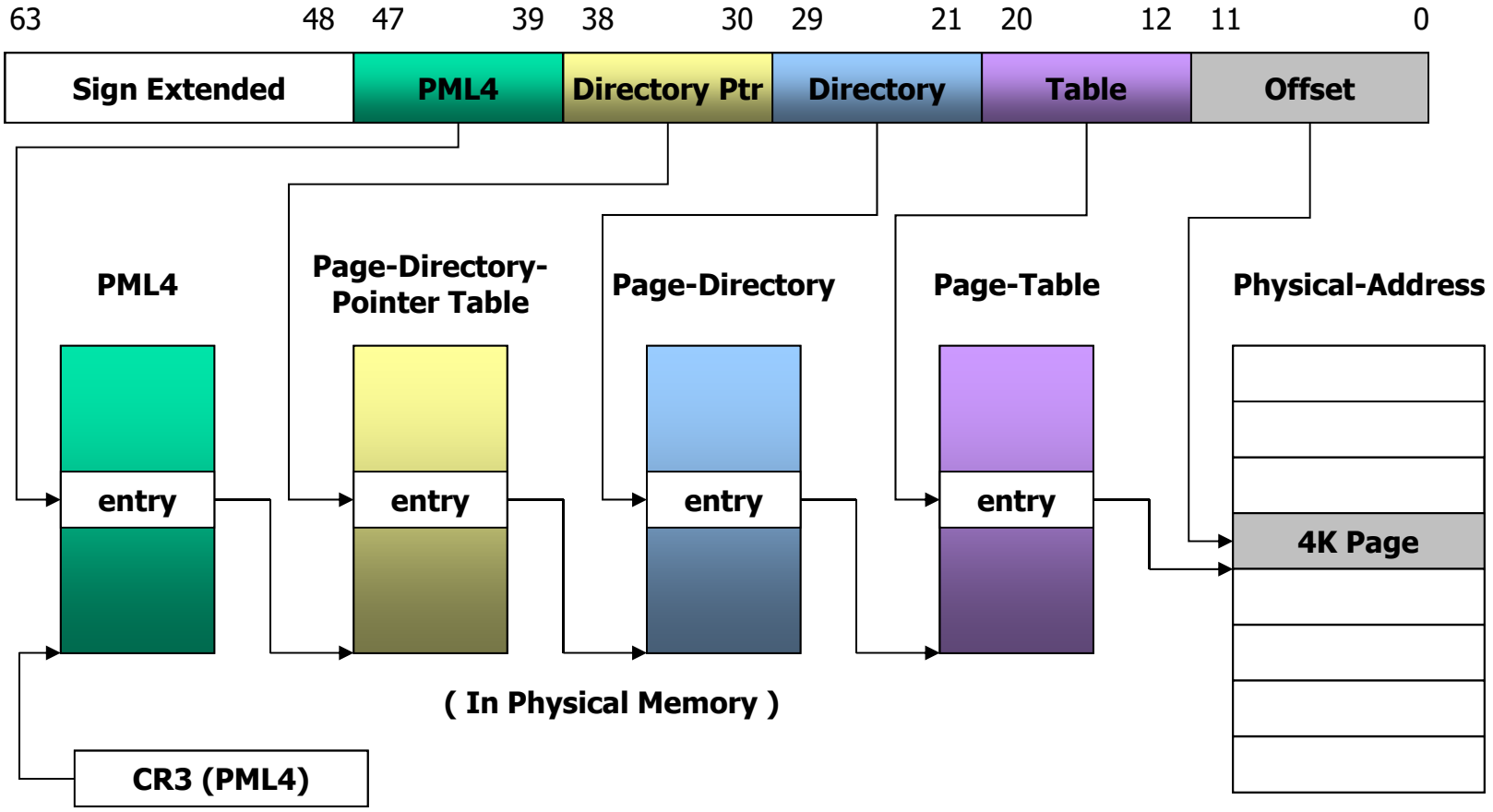
Memory Region



- ✓ pgd:
  - page directory address
- ✓ vm_prot:
  - read/write permissions for this area
- ✓ vm_flags
  - shared with other processes or private to this process

2.6.11부터 도입된 4단계 페이징(x86_64 기준)

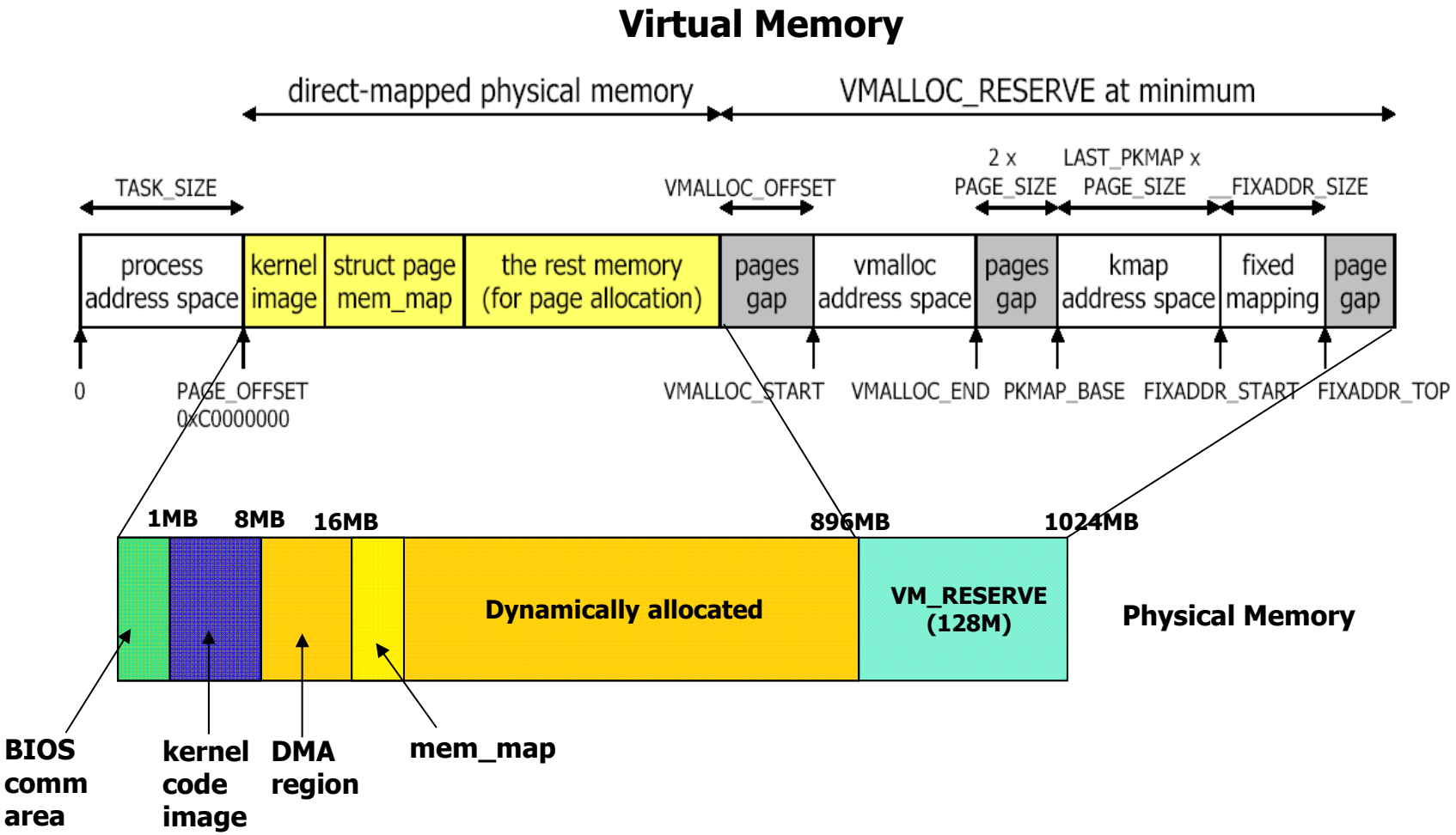
- 64bit address translation
  - ✓ 4-level translation



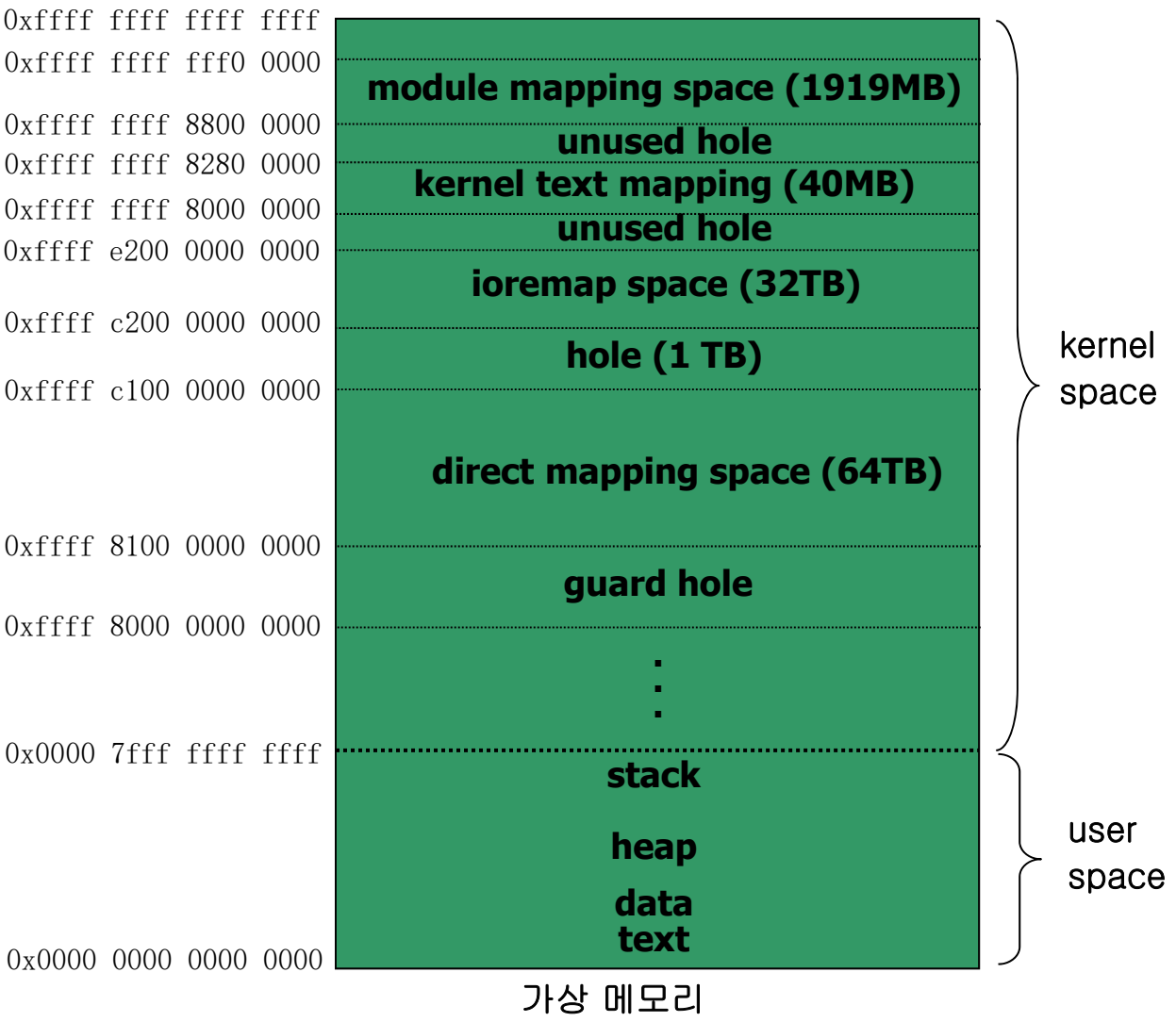
512 PML4 * 512 PDPTE * 512 PDE * 512 PTE = 2^{36} Page frames (256TB)

커널의 가상 주소 사용(32bit 처리기 기준)

- Kernel address space(32bit)

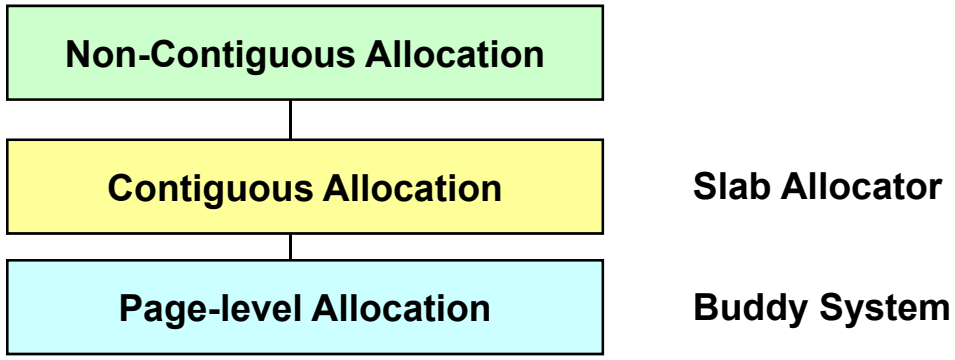


커널의 가상 주소 사용(64bit 처리기 기준)

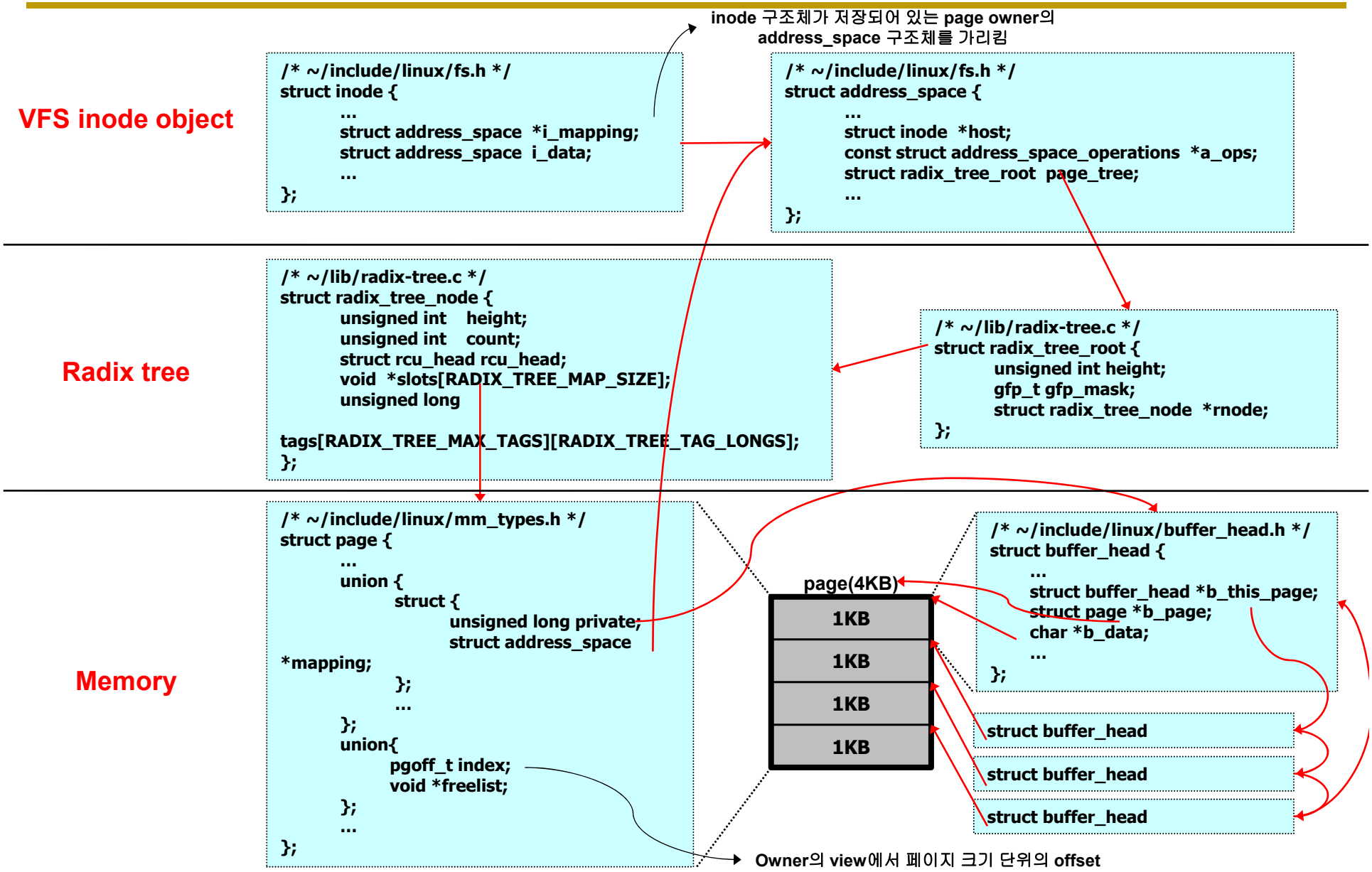


Linux 커널 메모리 관리 함수

- Interface
  - ✓ struct page * alloc_pages(unsigned int gfp_mask, unsigned int order)
    - 2^order 크기의 연속된 물리적 페이지들을 할당 후, 첫 페이지의 page 구조체 포인터를 리턴
  - ✓ void * page_address(struct page *page)
    - 매개변수로 넘긴 물리적 페이지가 현재 포함돼 있는 논리적 주소에 대한 포인터 반환
  - ✓ Wrapping Function
    - **Page-level allocator : get_free_page()**
      - 요청된 만큼의 연속된 메모리를 할당
    - **Kernel-level allocator**
      - 물리적으로 연속하며, 128KB이하 임의 길의의 메모리 할당 : **kmalloc()**
      - 물리적으로 비 연속적인 메모리 할당 : **vmalloc()**



Page Cache 관리 자료구조



Page Cache related Interfaces

- Radix Tree
  - ✓ find_get_page();
  - ✓ add_to_page_cache();
  - ✓ remove_from_page_cache();
  - ✓ read_cache_page(); // 메모리 상의 내용을 최신으로 유지

- Buffer Head
  - ✓ grow_buffers();
  - ✓ try_to_release_page();
  - ✓ __find_get_block(); //찾는 내용이 없으면 NULL return
  - ✓ __get_blk(); //찾는 내용이 없으면 이를 위한 자료구조 만들고 return
  - ✓ __bread(); //찾는 내용이 없으면 이를 위한 자료구조 만들고 필요시 I/O 수행

Page Cache Reclaim

- When?
  - ✓ The page cache gets too full and more pages are needed
  - ✓ Too many dirty pages
  - ✓ Too much time has elapsed since a page has stayed dirty
  - ✓ The flush request are invoked
- Parameters
  - ✓ Dirty background threshold
    - 보통 약 40%(/proc/sys/vm/dirty_ration)
  - ✓ Dirty writeback centisecs
    - 보통 500
- Pdflush daemon
  - ✓ background_writeout()
    - Flush 대상 dirty page 주기적인 탐색
  - ✓ wb_kupdate()
    - 너무 오래된 dirty 탐색
      - suber_block 은 5초
      - 일반 data page는 30초
- Evicting
  - ✓ submit_bh();
  - ✓ ll_rw_block(); //불연속 블록에 대한 I/O 처리