



**Xilinx Zynq FPGA,TI DSP,  
MCU 기반의  
프로그래밍 전문가 과정**

날 짜 : 2018 . 4. 10

강사 – Innova Lee(이상훈)

[gcccompil3r@gmail.com](mailto:gcccompil3r@gmail.com)

학생 – 정한별

[hanbulkr@gmail.com](mailto:hanbulkr@gmail.com)

## < 리눅스 커널 내부 구조\_ Chapter\_ 3>

### 1. 실시간 태스크 스케줄링( FIFO, RR , DEADLINE )

- CPU 를 어떤 태스크가 사용하도록 해줄 것인가? **공정**해야 여러 태스크들이 불평하지 않을 것이다.

- **스케줄러의 역할**이 **효율적**이어야 가장 **높은 처리율**을 낼 수 있다. **가장 급한 태스크를 한가한 태스크보다 먼저** 수행될 수 있도록 해준다면 보다 **높은 반응성**을 보일수 있다.

- 위의 3 가지 정책중에 deadline 정책을 주로 사용한다고 보면 된다.

- **실시간 태스크**는 우선순위 설정을 위해 task\_struct 구조체의 rt\_piriority 필드 사용.  
(0~99 까지의 우선순위), (100~139 는 일반 태스크)

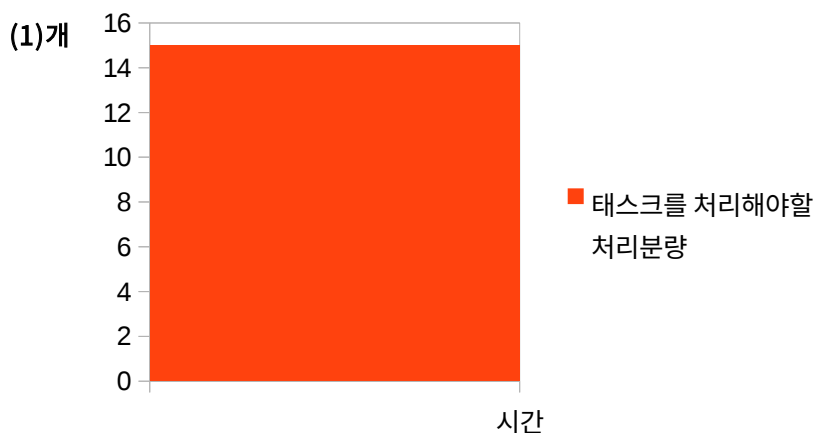
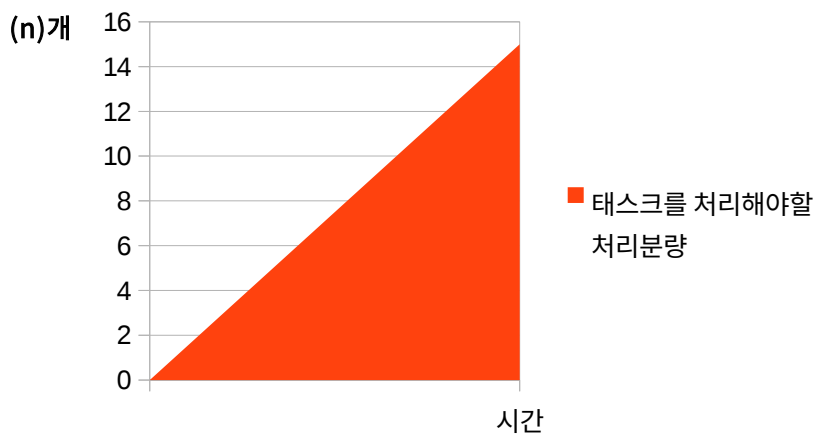
→**커널은 하이브리드 방식**을 가져서 여러가지 스케줄링을 한다.

- RR 방식은 번갈아가며 공정하게 동작하는 방식인데, 기본적으로 우선순위를 고려한 동작방식이다.  
( 동일 우선순위에 따라 time slice 기반으로 동작한다. 동일우선순위가 없을 때 , FIFO 와 동일 하다.)

- 시스템에 존재하는 모든 태스크는 tasklist 라는 이중 연결 리스트에 연결되어 시스템내 모든 task 접근 가능.

- **하지만** 위와 같은 연결은 문제가 있다. **태스크의 개수가 늘어나면** 그만큼 **스케줄링에 걸리는 시간도 선형적으로** 증가하며 ( **$O(n)$ 의 시간복잡도**), 따라 스케줄링 **시간의 예측이 불가능** 하다.

#### $O(n)$ 의 시간 복잡도



1. 위의 단점을 해결 하기 위해서, 태스크는 0~99 의 우선순위 레벨에 맞는 비트맵을 생성한다.
2. 비트맵에서 그 태스크의 우선순위에 해당하는 비트를 1 로( 나중에 ‘&’ 연산으로 check) set 한 뒤 해당하는 우선순위 큐에 삽입한다.
3. 스케줄링 하는 시점이 되면 커널은 비트맵에서 가장 처음으로 set(우선순위높은) 된 비트를 찾아 선택한다.
4. 비트맵에서 최우선 비트를 찾아내는 것은 상수시간 안에 가능함으로 고정시간 내 완료된다.

- 비트맵을 가지고 queue 에 넣는게 hash table 이다.

## \*deadline 정책

기존 리눅스의 실시간 태스크 스케줄링 정책은 우선순위에 기반하여 스케줄링 대상을 선정하는데 반해 , deadline 정책은 잠시 후 설명될 **deadline 이 가장 가까운 (즉 가장 급한) 태스크를 스케줄링 대상으로 선정한다.**

예를 들어 동영상을 재생하는 태스크가 수행중 이라고 가정하자.

초당 30 프레임을 디코딩 하여 화면에 출력하면 태스크는 1 초당 30 번 씩 ‘해야하는 일’ 을 가지고 있는 것이다. 이 때, (1 초/ 30)보다는 적은 시간내에 수행해야할 ‘작업량’ 을 가질 수 있음을 의미한다.

여기서 ‘초당 30 회’ 는 **period** 라고 부르고 ‘작업량’ 은 **runtime**, ‘완료시간’ 을 **DEADLINE** 이라고 한다.

이 DEADLINE 정책을 사용하는 각 태스크들은 **deadline 이용하여 RBTREE 에 정렬**되어 있으며, 스케줄러가 호출되면 **가장 가까운 deadline** 을 가지는 태스크를 **스케줄링 대상으로 선정**한다. 즉 **우선순위가 상관없다.** (영상, 음성, 스트리밍 등, **제약 시간을 가지는** 수많은 응용들에 용이하다.)

\*커널 내에서 ( vi -t 로 들어간 직후, kernel 디렉토리에서 ) driving 하며 찾을 때, 명령어 **:cs find 0 ‘파일이름’**

## 2. 일반 태스크 스케줄링( CFS )

-- 가상적으로 공평하게 , 실제시간으로 공평하게 한다. --

**완벽하게 공정한 스케줄링**, **cpu 사용시간에서 공정한 분배**를 의미한다. A, B 두 태스크가 있다면 사용시간이 1:1 로 같아야 한다.

시간이 항상 1:1 일 수는 없다. **CFS 는 정해진 ‘시간단위’ 로 봤을 때, 공정한 시간을 할당하는 것을 목표**한다.

1 초를 시간단위로 하면 0.5 초는 A 태스크가, 0.5 초는 B 태스크가 동작 함으로써 A 와 B 의 사용시간이 1 초간 1:1 이 되도록 하는 것이다.

즉 런큐에 n 개의 태스크가 존재하면 ‘시간단위’ 를 n 으로 나누어 n 개의 태스크에게 할당해 주면 된다.

\*여기서 우선순위 높은 태스크게 가중치를 둔다면 우선순위가 높은쪽에 가중치를 높인다.

- 이를위해 리눅스는 **vruntime** 을 만들었다.

(**우선순위가 높은 것은 시간을 많이 가져야** 하는데 그래서 **곱해지는 가중치가 적어야 한다.**)

**$vruntime += physicalruntime * weight(0) / weight(curr)$**

( **weight(curr)**: 현재 태스크의 weight 값을 의미, **weight(0)** : 기본 단위인 1024 )

- 스케줄링이 되는 대상을 빠르게 골라내는 방법은 가장 작은 vruntime 값을 가지는 태스크가 가장 과거에 cpu를 사용 했음을 의미 한다.

따라서, RBTree 는 맨 왼쪽에 작은 값이 오는 규칙을 이용해 금방 찾아 낼 수 있다.

- 위의 다음 문제로 vruntime 값이 항상 작은 값으로 스케줄링 된다면 너무 자주 스케줄링이 발생하지 않을까? Context switching 을 줄이려면 time slice 를 최대한 사용한다.

- 스케줄러는 언제 호출될까?(nice system call)

1. schedule() 함수를 호출하는 방법

2. thread\_info 구조체 내부에 존재하는 flags 필드중 need\_resched 라는 필드를 설정.

- ( 그룹: 특정 사용자 )간의 스케줄링은 공정하게 그룹 1 초, 특정사용자 1 초 식으로 cpu 가 공정하게 분배를 한다.

## Chapter 4

### 1. 물리 메모리 관리 기법과 가상 메모리

- 사용자는 물리적으로 존재하는 메모리 보다 더 많은 양의 메모리를 필요로함, 그래서 많은 기법들이 개발되었다. 그 중에서 가장 성공적이고 지금도 대중적인 방법이 가상 메모리 이다.

- 어차피 가상적으로 주는 공간이라면 되도록 많은 공간을 준다. 32 비트의 cpu 의 경우 4GB 의 가상 주소 공간을 사용자에게 제공 한다. 64 비트의 경우는 Exa Byte 의 크기를 할당 할 수 있다. (너무큼)

- 한가지 주의할 점은 4GB 모두 사용자에게 제공하는 것이 아니다. 그저 개념적으로 제공되는 가상 공간이다. 실제로는 사용자가 필요한 만큼의 물리 메모리를 제공한다. (커널에 sys\_exceve 가 물리메모리로 가져옴)

- 필요한 만큼의 물리메모리 사용으로 많은 태스크가 동시에 사용 가능, 커널이 메모리 배치를 알아서 하기 때문에 배치 정책이 불필요 하다.

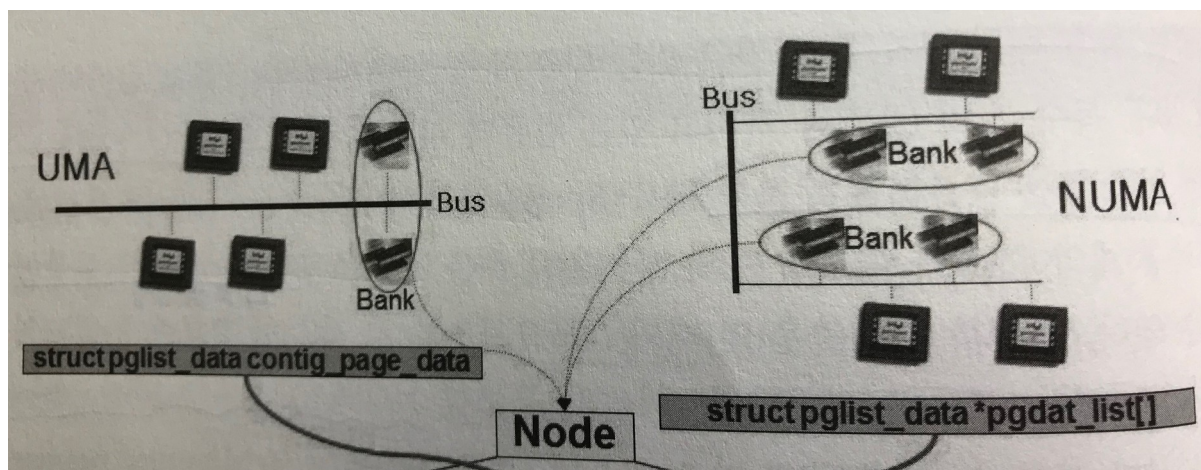
## 2. 물리 메모리 관리 자료 구조

- 처음 부팅시 전체 물리 메모리에 대한 정보를 가지고 온다.
- 물리적 자원이 리눅스에서 어떻게 표현되는지는 (paging, slab, uma, numa 등이 있다.)
- 요즘 인기종 시스템(NUMA)은 CPU 를 몇 개의 그룹으로 나누어 메모리 입출력 버스간 공유(SMP)에서 병목 현상이 일어나는 것을 줄이기 위해 사용하는 기법이다.
- UMA 구조는 BANK 1 개 가지고 있다. NUMA 구조는 BANK 2 개 이상으로 CPU 에서 어떤 메모리에 접근하느냐에 따라 속도가 달라진다, 되도록 가까운 메모리에 접근해야한다.

### 2-1 Node

- 접근속도가 같은 메모리의 집합을 뱅크(bank)라 부른다. 리눅스에서 bank 를 표현하는 구조가 'node' 이다.
- UMA 의 경우 한개의 node 가 존재할 것이다. 이 node 는 contig\_pagedata 를 통해 접근 가능하다.
- NUMA 의 경우 복수개의 node 가 존재할 것이다. 복수개 node 는 list 를 통해 관리되며 pglist\_data 라는 배열을 통해 접근이 가능하다.
- 위 contig\_pagedata , pglist\_data 는 물리메모리 실제양, 물리메모리 �핑 위치 등의 정의 되어 있다.
- 리눅스가 물리메모리 할당 요청을 받으면, 되도록 할당 요청한 태스크(cache 를 쓴단 의미)가 수행되고 있는 cpu 와 가까운 node 에서 메모리를 할당하려 한다.

\*cache 는 자주 쓰는것들이나 빨리 써야 하는 것들을 저장하는 공간이다. 빠르게 읽고 쓰기가 가능하다.



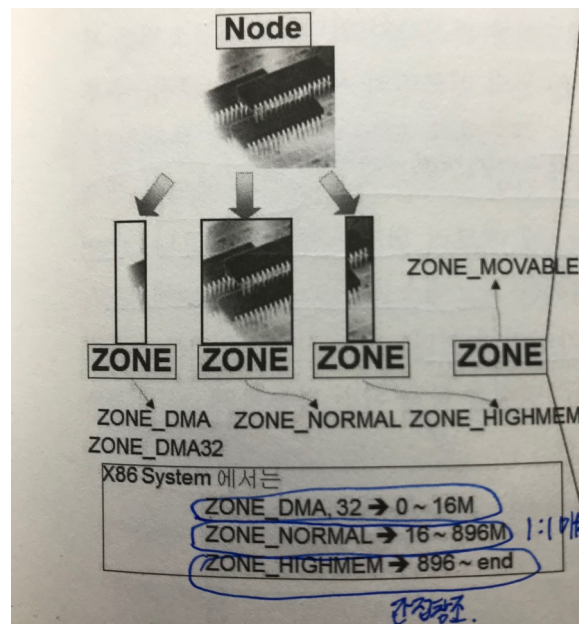
## 2-2 Zone

- node 안에 존재 하고 있는 물리 메모리는 모두 어떠한 용도로도 사용될 수 있어야 한다. 그 결과 node 에 존재 하는 물리메모리중 16MB 이하 부분을 좀 특별하게 관리 한다.

- 이를 위해 **node** 의 일부분을 따로 관리 할 수 있도록 **자료구조**를 만들어 놓았다.

- zone 은 동일한 속성을 가지나, 다른 zone 의 메모리와는 별도로 관리되어야 하는 메모리의 집합이다.

- ZONE\_DMA (direct memory access) 0~16M → **cpu 에 거치지 않고 memory 에 접근 한다.**  
(서버,비디오, 사운드→ V4L2 , ALSA(코덱만드는 기술))



- ZONE\_NORMAL (16~ 896M) → 1 대 1 로 매칭해 저장하는 공간.

- ZONE\_HIGHMEM (896 ~ end) → 동적으로 연결하여 사용할 수 있는 **간접 참조** 구조

- **64bit 운영 체제**는 공간이 워낙 많기 때문에 위의 3 가지 형태의 존이 다 있지 않고 1:1 매칭만으로도 가능하다.

- zone 구조체는 **free\_area** 를 가지고 있는데 **버디 할당자**가 사용하는 영역이다.

- zone 에는 watermark , vm\_stat 을 통해 빈 공간이 부족시 적절한 메모리를 해제를 한다.

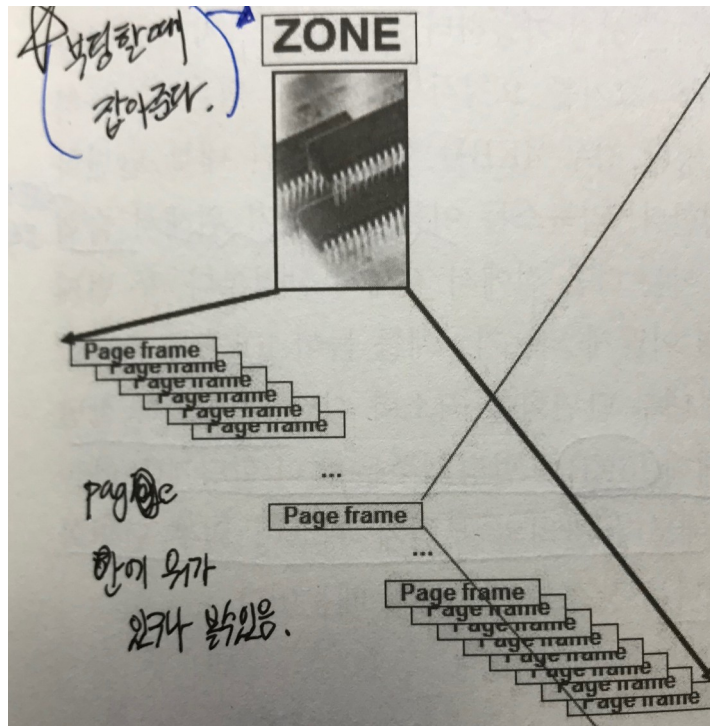
1. watermark : 워터마크가 표시된 곳을 넘어가면 해제 하는 옵션.

2. vm\_stat : 현재 상태를 가지고 있다.



## 2-3 Page frame

- 물리 메모리의 최소 단위 이다. **page** 는 이 **page\_frame** 을 관리 하는 소프트웨어 이다.
- page 구조체는 **kmem\_cache \*slab\_cache** 라는 **슬랩 할당자**를 가지고 있다.
- **page\_frame** 하나당 **page** 구조체가 존재한다.
- **page** 구조체는 4kbyte 가 나오면 안된다. 다른정보도 저장해야 하니까. 더 작아야 한다.
- 부팅되는 순간에 구축이 된다. 이 위치는 **mem\_map** 이라는 전역 배열을 통해 접근 된다.(node→ **pglist\_data**)  
(**mem\_map** 은 디스 컨티그가 없으면 노드는 '0' 하나다.)



## 3 Buddy 와 Slab

- 사용자가 1byte 가 필요시 1byte 단위로 할당해 버리면 이를 관리하는 메타데이터가 넘어 서기 때문에 1byte 보다 큰 양을 할당해야 한다.
- **페이지 프레임 단위 (4kB)** 로 할당 하도록 결정 한 방식이 **Buddy** 이다. ( $4KB \times 2^n$  방식으로 동작)
- 만약 **작은 단위로 크기를 요청** 해줄 경우 (30byte 나 60 byte 정도) 4KB 를 할당 하면 내부 단편화가 올 수 있다. 이를 해결 하기 위해 **Slab 할당자**를 이용한다. **4KB 에서  $2^n$  으로 쪼개서 사용하는 방식**이다.

### 3-1 Buddy 버디 할당자( 소프트웨어 로직 )

- buddy 는 zone 당 하나씩 존재 한다.

- zone 구조체에 존재하는 **free\_area[]** 배열을 통해 구축 되며 그 안에는 **free\_list** 와 **nr\_free** 라는 필드를 갖는다.

1. free\_list : 비어있는 리스트

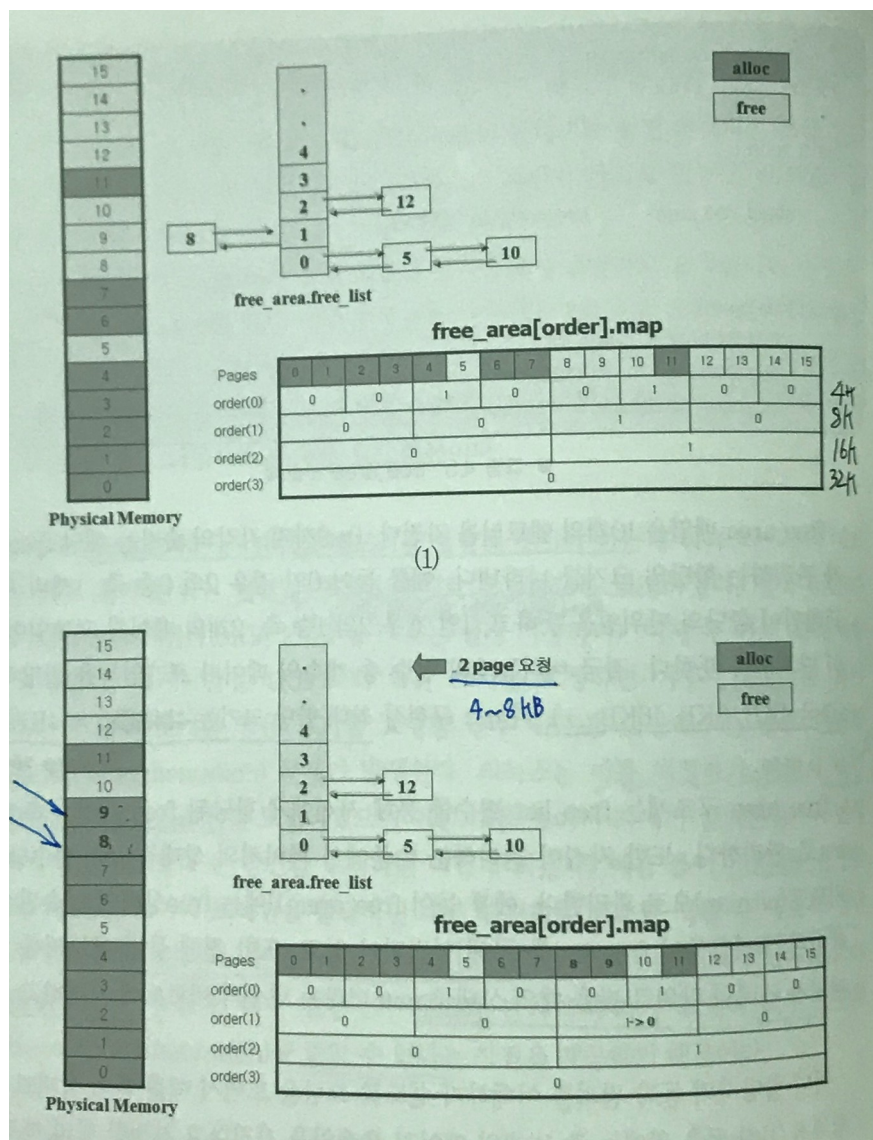
2. nr\_free : 애가 몇개 비어 있나 알려준다. (비사용중인 페이지 프레임 개수)

- 리눅스 구현상 최대 할당 크기는 ( $2^9 \times 4KB$ ) 이다.

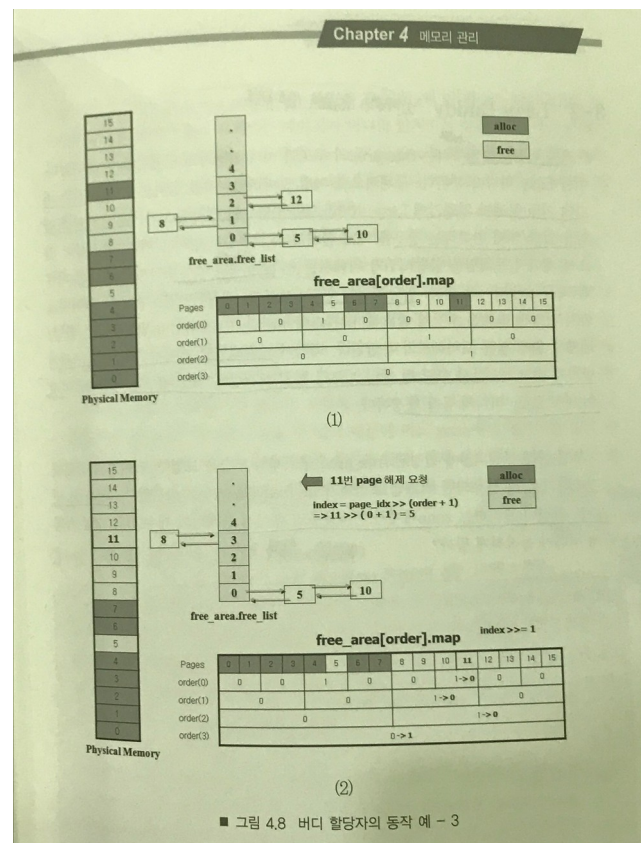
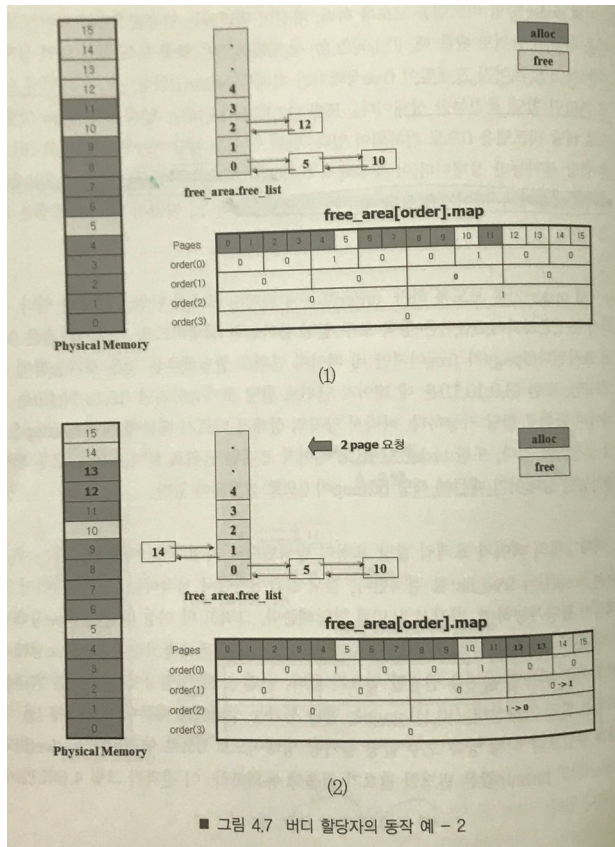
- 외부 단편화 최소화 해주었다.

- 몇 K 가 나오든 나누어 주기가 편하다.

- 한번 하면 두번 다시 페이지를 못한다. 하지만 덕분에 빠르다.







## 3-2 Lazy Buddy ( 소프트웨어 로직 )

- 기존 버디의 문제점이 있어서 만들게 되었다.

- 한 페이지 프레임을 할당/해제 반복하는 경우.

페이지 할당을 위해 페이지를 **slab(조개서)** 할당해 주어야 한다. (비트맵 수정해야함) 그리고 **할당을 해제** 하면 다시 큰 페이지로 **합쳐** 지면서 (비트맵이 수정되어야함) 다시 큰 페이지로 **돌아온다**.

이 것이 **반복**이 되면서 할당 해제를 위해 오버 헤드가 동반된다.

→ 할당된 페이지를 구태여 바로 다시 합치지 말고 곧 다시 할당 되는 것을 생각해 작업을 뒤로 미룬다.

→ zone 에 있는 watermark 를 페이지 가능 수를 비교 후 zone 메모리가 충분 시 병합을 뒤로 미룬다. 가용메모리가 부족 시에 해제 한다.

\* 버디 할당 관련 정보 : cat /proc/buddyinfo

### 3-3 슬랩 할당자 ( 소프트웨어 로직 )

- 내부 단편화를 없애 주었다.

- 우리는 자료구조에서 32byte 씩 했다 그래서 **느려졌다**. 새로 쪼개야 하기 때문에...

1. 페이지 프레임 을 받아서 이 공간을 나눈다.
2. 그리고 버디할당자로 부터 받지 말고 **나뉜 공간으로 받는다**.
3. 해제시에는 버디로 반납하는게 아닌 **미리 할당 받은 공간에 그대로 가지고** 있다.
4. 일종의 **캐시**로 사용하는 것이다.

→ 이런 **캐시 집합**을 통해 **메모리**를 관리하는 정책을 **slab 할당자**라고 한다.

(kmem\_cache)→(kmem\_cache\_node) \*slab\_cache

- kmem\_cache 를 위한 공간을 할당받아 다양한 캐시를 생성할 수 있게 되는 것이다.

- 슬랩 할당자는 외부 인터페이스 함수로 kcalloc()/kfree()를 제공 한다.

kcalloc() : 메모리를 순차적으로 배치 **cache 의 locality (지역성)**,  
최대 128k~4MB 할당된 공간은 **연속(순차적 배치) 특징으로 빠르다**.

### 4 가상 메모리 관리 기법

태스크당 하나인 task\_struct 와 태스크 의 가상 주소 공간의 관계에 대해 살펴 보자.

태스크는 자신의 **고유한 가상 메모리**를 갖는다. 따라서 커널은 **태스크의 가상 메모리가 어디에 존재하는 지** 관리를 해야 한다.  
즉, 어디에 **text 영역**이 있고, 어디에 **data 영역**이 있는지, 그리고 **어는 영역이 사용 중이며 어느 영역이 사용 가능한지** 등등의 정보를 알아야 한다.

- task 마다 ( task\_struct ) → ( mm\_struct )라는 구조를 가지고 있고 여기서 메모를 관리하고 있다.

- \*\* 메모리가 관리하는 주 대상은 ‘세그먼트’ , ‘page’ , ‘page\_fault\_handler’ 가 있다.

- 메모리 공간중 **같은 특성의 연속적인 영역**을 ‘region’ = ‘세그먼트’ 인 것이다.

- region 을 vm\_area\_struct 라는 자료구조를 통해 관리한다. → **레드 블랙 트리**로 연결해 효율적 관리를 한다.

( red black tree 의 시작을 나타낸 변수는 mm\_rb, 최근 접근한 vm area struct를 나타낸 변수 mmap\_cache)  
(mmap → 스티키 비트)

- 주소 변환을 위한 **페이지 디렉터리의 시작점 주소**를 pgd 라고 한다. ( 가상메모리→ 물리메모리 )

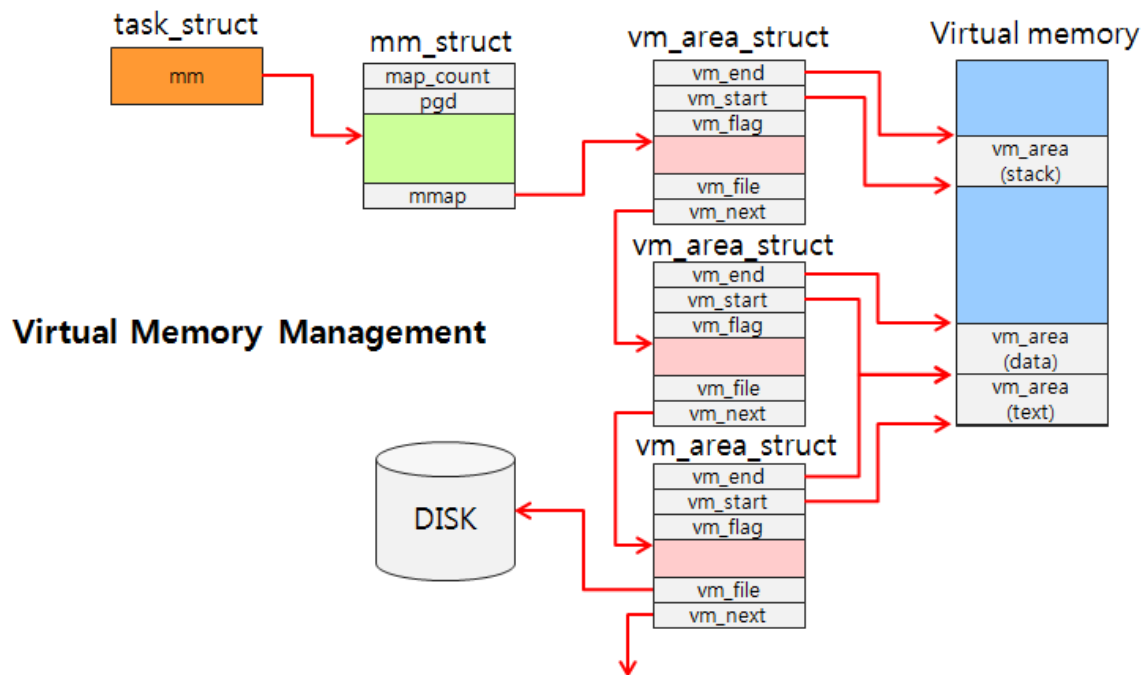
- vm\_area\_struct 에는 **실제 실행 파일의 어느 위치에 있는지에** 대한 정보를 vm\_file 와 vm\_offset 변수로 관리 한다.

\*\*위 변수들은 **페이지 폴트가 발생시**에 **파일에 어느 부분을 읽어야 하는지 결정**할 때 사용한다.

- pgd : page directory address

- vm\_port : read/write permissions for this area

- vm\_flag : shared with other processes or private to this process



第七城市 www.7thcity.com

- 그림에서 보듯이 **vm\_area** 는 공통의 속성을 갖는 페이지 들이 모여서 이루어지고 **vm\_area\_struct** 라는 자료 구조를 통해 관리 당한다.
- **page fault** 발생시에 커널에 어디가 문제가 인지 알수 있으면 **page\_fault\_handler** 에 의해서 **페이지를 할당** 후 동작한다.
- 태스크 안에서 여러 **vm\_area\_struct** 는 겹치지 않고 **속성이 같다면 하나로 묶어 통합적으로 관리**할 수 있다. (연결 리스트가 필요하지 않아서 빨라진다.)

## 5. 가상 메모리와 물리 메모리의 연결 및 변환

- 태스크를 하나 생성하고 태스크에게 가상 주소 공간을 제공해 주고, 메모리의 일부를 할당해 준다(**page**), 태스크가 원하는 디스크 상의 내용을 읽어 물리 메모리에 올려놓고(**sys\_execve**) 물리 메모리의 실제 주소와 태스크의 가상 주소 공간을 연결 한다.
- **sys\_execve()** : 메모리를 적재하는 일을 한다.
- 커널 공간도 텍스트와 데이터, 그리고 스택으로 구분되어 있다. 스택의 **s1** 주소는 3G 이고 3G-4KB 아래로 쌓인다.
- **sys\_execve()** 시스템 호출은 우선 **free** 한 **페이지 프레임**들을 할당 받는다.(**demand on page**) 그리고 메모리 적재.
- 컴파일러가 코드를 적재할지 먼저 써주고 **sys\_execve()**가 적재해주는 작업을 한다.
- **페이지 테이블**은 **가상 주소를 물리주소로 변환** 하는 주소 변환 정보를 기록한 **테이블** 이다.

- page table 의 엔트리에서 NP(not present)가 있으면 메모리가 적재 되어 있지 않아서 페이지 폴트가 발생하게 된다.  
(mm\_struct→vm\_struct → flags 의 권한→ 커널이면 페이지를 할당 한다.)

페이지 폴트가 발생 후 페이지 폴트 핸들러에 의해 demand on paging 을 하여 페이지 프레임을 할당,읽어다 놓는다.

- mm\_struct 라는 자료 구조에서 pgd 라는 필드는 가상주소를 물리주소로 변환을 위한 시작 페이지 프레임 번호를 가진다.

- PGD → PMD → PTE 순으로 인덱싱 한다. (각 시작점을 offset 하고 있다.)

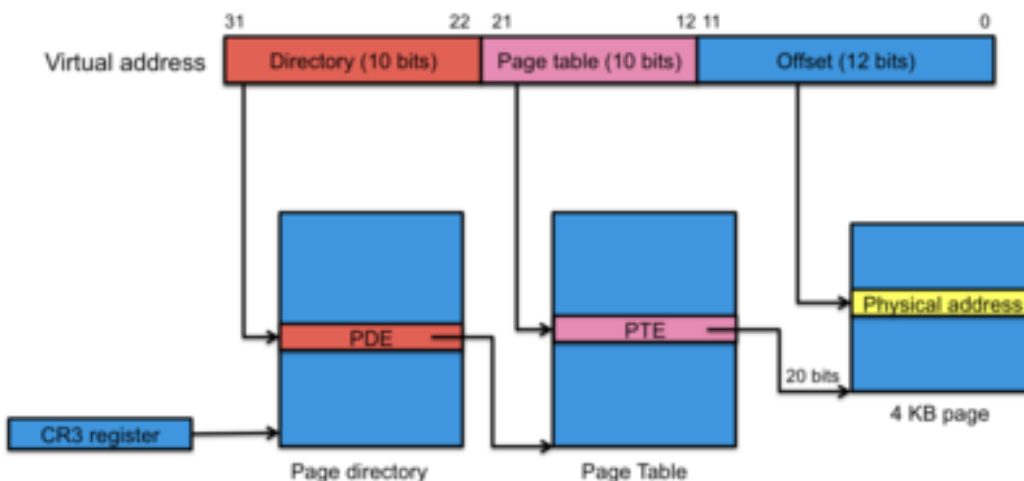
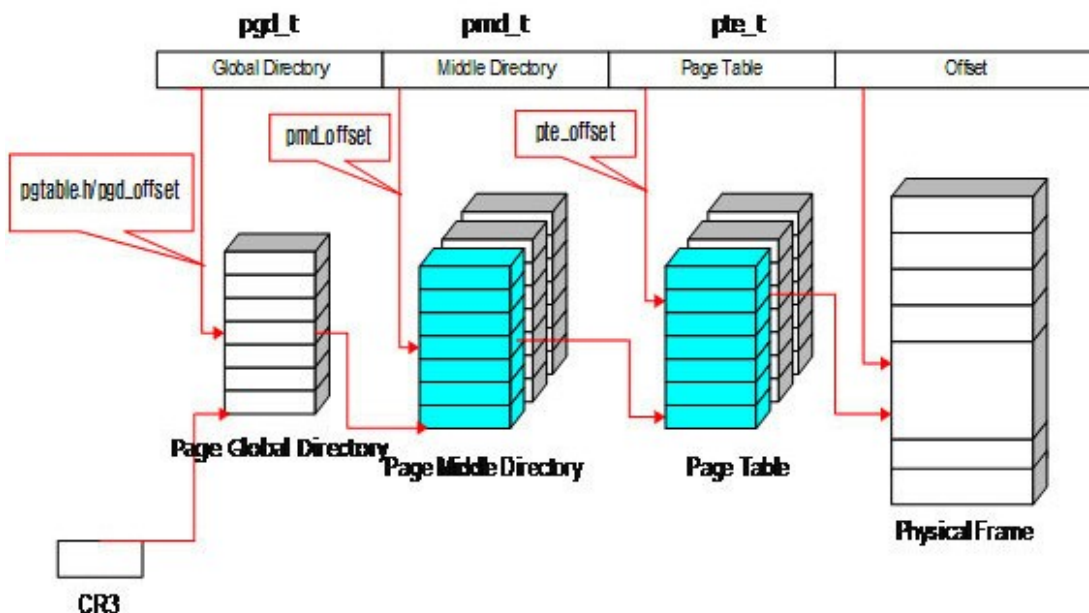
- cr3 는 cpu 에 있는 진짜 레지스터 이다.

- 버디나 슬랩 할당자를 통해 받은 메모리 공간은 늘 넉넉한건 아니다. vmalloc()을 사용,동적할당 받아 쓰면 된다.

### -가상메모리 장점

1. 물리 메모리의 관계 없이 큰 주소 공간을 프로그래머에게 제공해 준다.
2. 모든 페이지를 적재할 필요 없이 page fault→ page fault handler→ demand on page 에 의해 관리한다.
3. 여러 태스크가 특정 영역을 공유하고 싶을 때(쉐어드 메모리) 단지 페이지 테이블에서 같은 프레임을 가리켜 공유가 된다.

### 64 bit linear address



### 32bit 운영체제 의 페이지징



- MMU 는 TLB 를 가지고 이미 변환된 주소(paging)를 캐시해 준다.

