# Xilinx Zynq FPGA, TI DSP, MCU 기반의 프로그래밍 및 회로 설계 전문가 과정

# #34

강사 : Innova Lee(이 상훈)

학생 : 김 시윤

## 1.배운내용 복습.

---

## 실시간 태스크 스케줄링 (FIFO, RR and DEADLINE)

---

CPU를 어떤 태스크가 사용하도록 해줄 것인가?
-> 스케줄러 – 시피유를 누가 획득할 것인가를 관리.

공정해야 여러 태스크들이 불평하지 않을것!

효율적이어야 가장 높은 처리율을 낼 수 있을것!

가장 급한 태스크를 한가한 태스크보다 먼저 수행될 수 있도록 해준다.
-> 우선순위 급한게 우선순위가 높다!

어떤 기준을 근거하여 태스크를 골라 낼 것인가?
task_struct 구조체에 policy , prio, rt_priority 필드가 존재한다.

**policy** -> 어떤 스케줄링 정책을 사용할지를 나타낸다.
(FIFO , RR , DEADLINE)

**prio** -> prio는 우선순위를 나타내며 동적이다.
**rt_prio**-> 실시간 우선순위를 나타내며 정적이다.

리눅스의 태스크는 실시간 태스크와 일반 태스크로 나뉘며, 실시간 태스크를 위해 3개, 일반 태스크를 위해 3개 총 6개의 스케줄링 정책이 존재한다.

실시간 태스크를 위해서는 SCHED_FIFO , SCHED_RR , SCHED_DEADLINE 정책이 사용되고 일반 태스크를 위해서는 SCHED_NORMAL 정책이 사용된다. 더불어 중요하지 않은 일을 수행하는 태스크가 cpu를 점유하는 것을 막기 위해 가장 낮은 우선순위로 스케줄링되는 SCHED_IDLE 정책, 사용자와의 상호작용이 없는 태스크를 위한 SCHED_BATCH 정책이 존재한다.

실시간 태스크는 우선순위 설정을 위해 **task_struct 구조체의 rt_priority** 필드를 사용한다. **rt_priority는 0~99까지의 우선순위를 가질수 있다.**

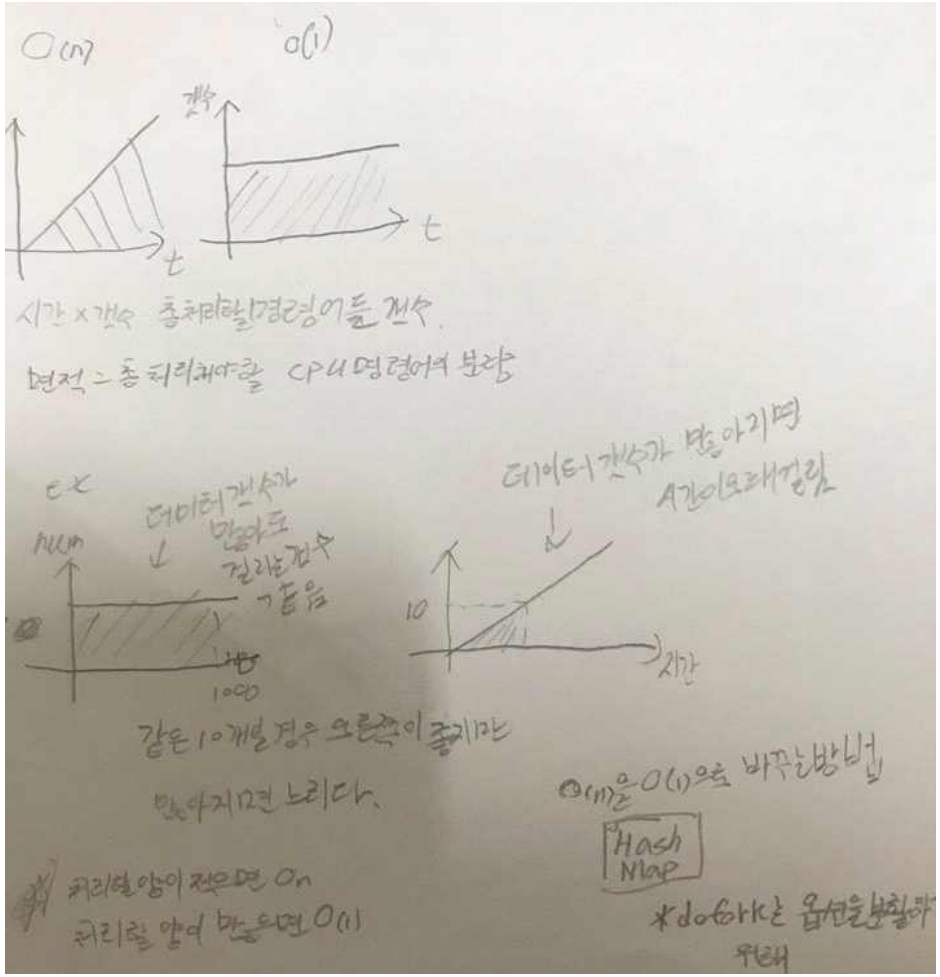
SCHED_RR –> 태스크가 수행을 종료하거나 , 스스로 중지하거나 , 혹은 자신의 타임슬라이스를 다 쓸때까지 CPU를 사용한다. 즉 우선순위가 같은 태스크가 여러개 존재할 경우 똑같은 시간을 할당한다.

SCHED_FIFO -> 우선순위를 가지는 태스크가 존재하지 않는경우! 먼저 들어온 태스크 순으로 동작한다.

실시간 정책을 사용하는 태스크는 고정 우선순위를 가지게 된다. 따라서 우선순위가 높은 태스크가 낮은 태스크보다 먼저 수행될 것을 보장한다.

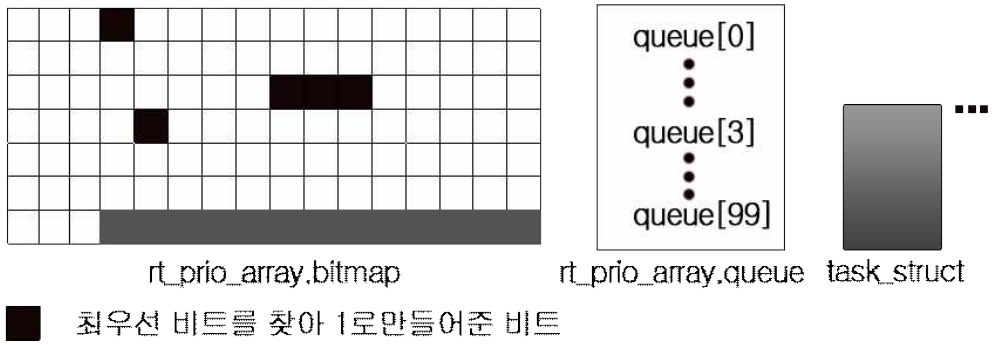
우선순위가 높은 태스크를 찾는방법은 아래와 같다.
모든 태스크들은 tasklist라는 이중 연결리스트로 연결되어있으므로, 이 리스트를 이용하면 시스템내의 모든 테스크를 접근하는 것이 가능하다.
(* tasklist 에는 next 와 prev 가 존재하며 이전에 task와 다음에올 task를 알 수 있다.)

하지만 이러한 방식에는 문제가 있다. 태스크의 개수가 늘어나면 그만큼 스케줄링에 걸리는 시간도 선형적으로 증가하게 된다.
(O(n))




On 과 O1 의 차이점 및 개념 hash map

태스크들이 가질수 있는 모든 우선순위 레벨(0-99)를 표현할수 있는 비트맵을 준비한다. 태스크가 생성되면 비트맵에서 그 태스크의 우선순위에 해당하는 비트를 1로 set한뒤, 태스크의 우선순위에 해당되는 큐에 삽입된다.




최우선 비트를 1로 만들어 앤드 비트연산 후 queue에 넣는다.
queue0은 우선순위가 제일 높은녀석들이 들어있다.
queue3 은 우선순위 4번째인 녀석들이 들어있다.
각 queue배열에는 index 번호의 우선순위를 갖는 태스크들을 저장한다.
이런식으로 관리하는게 hash라고 한다.

DEADLINE

DEADLINE 정책은 잘 알려진 실시간 태스크 스케줄링 기법 중 하나인 EDF 알고리즘을 구현한 것이다. 가장 가까운 ( 가장 급한) 태스크를 스케줄링 대상으로 선정한다.
모니터 그리기라는 작업을 해야하는 프로세스가 존재할 때
30번에 한번 모니터에 그림을 그려주는 일을 해야할 때 0.033 초 안에 모니터 그리기를 처리해 주어야한다. 실제 그림 그려지는 시간이 0.01초라 할 때 0.023초의 여유시간을 갖고 context switching 의 시간이 0.003초라 가정하면 실질적인 여유시간은 0.02초이다. 이 여유시간동안 다른 프로세스의 작업을 수행하는게 DEADLINE 방식이다.
deadline 방식은 RBtree에 정렬되어있다. ( thread의 삽입과 삭제가 빈번하기 때문) deadline은 struct dl_rq에서 관리한다.

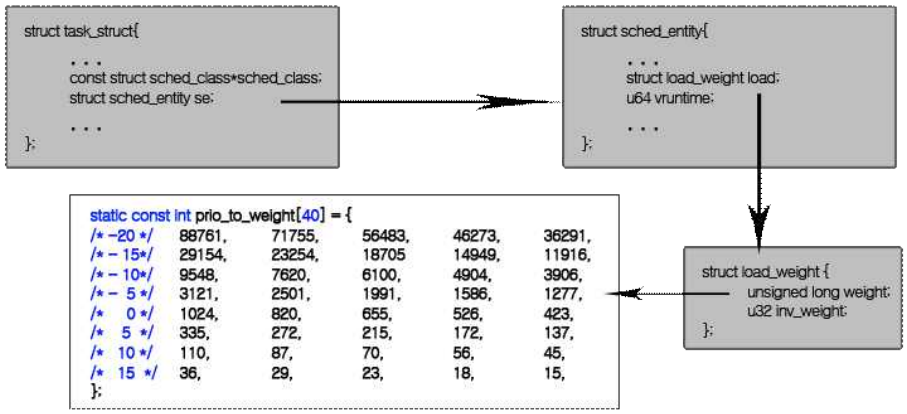
## 일반 태스크 스케줄링 (CFS)

리눅스가 일반 태스크를 위해 사용하고 있는 스케줄링 기법은 CFS라 불린다. CFS는 CPU 사용시간의 고정한 분배를 의미한다. A와 B 두 개의 태스크가 수행 중이라면, A와 B의 시피유의 가상시간이 1:1로 같아야한다.

Run queue 에 N개의 태스크가 존재한다면, N개의 똑같은 가상시간을 나누어 준다. 그리고 태스크의 우선순위에 따라 우선순위가 높은 태스크에게 가중치를 두어 좀더 긴 시간을 CPU를 사용할 수 있도록 해준다.

가상시간 = vruntime

일반 태스크 사용자 수준에서 볼 때 **–20 ~0 ~ 19** 사이의 우선순위를 갖게되며, 이 값은 커널 내부적으로 (priority + 120)으로 변환된다.
따라서 **실제 태스크의 우선순위는 100~139**에 해당되며, 항상 실시간 태스크의 우선순위보다 낮은 것이 보장된다.




위 그림에서 배열의 시작과 끝은 –20부터 19까지이다.(가중치를 나타내준다)

$$vruntime \mathrel{+}= physicalruntime \times \frac{weight_0}{weight_{curr}}$$

이때 weight curr은 현재 태스크의 weight값, weight0은 우선순위0에 해당하는 weight값을 의미한다.
예를 들어 현재 수행중인 태스크의 우선순위가 –20(88761)이고 1초(physicalruntime)rks 수행되었다면 vruntime은 아래와 같이 계산된다.
weight0은 0번(1024)

$$vruntime = vruntime + (1 \times \frac{1024}{88761})$$

즉 vruntime + 0.0115366 이므로 0.0115366초가 가중된다는걸 알 수 있다.

다시한번 정리하면 우선순위가 높은 태스크의 경우 좀더 긴시간 CPU를 사용할 수 있도록 하고 우선순위가 낮은 태스크인 경우 좀더 짧은 시간동안 CPU를 사용할 수 있도록 관리하는 것을 의미한다.

스케줄링의 대상이 되는 태스크를 어떻게 빠르게 골라낼 것이냐 하는 문제가 있다. 가장 작은 vruntime을 가지는 태스크가 가장 과거에 CPU를 사용했음을 의미한다. 따라서 리눅스는 이런 태스크를 다음번 스케줄링의 대상으로 선정함으로써 공정한 스케줄링을 수행하며 그 관리는 RBTree로 한다.
RBTree의 왼쪽이 없을 때 까지 찾으면 다음에 스케줄링 할 태스크를 찾을수 있다. 이것을 위로 올려 로테이트 시켜주면 삽입삭제가 활발하게 일어나므로 RBTree가 자료관리에 최적화임을 한번더 상기시켜준다.

한가지 문제가 더 남아있다. 항상 가장 작은 vruntime값을 가지는 태스크가 스케줄링 된다면 너무 자주 스케줄링이 발생된다. 그래서 빈번한 스케줄링을 막기 위해 스케줄링간 최소 지연 시간이 정의되어있고 그 시간은 context switching 시간보다 길다.
그 최소지연 시간은 __sched_period()함수에서 계산된다.

스케줄러는 언제 어떻게 호출될까?
직접적으로는 schedule() 함수를 호출(nice systemcall)하는 방법이 있고,
현재 수행되고 있는 태스크의 thread_info 구조체 내부에 존재하는 flags중 need resched라는 필드를 설정하는 방법이 있다.

thread_union -> thread_info -> flags-> need resched

마지막 CFS와 관련된 사항 한가지를 보면
철수가 99개의 task를 생성했고 영희가 1개의 task를 생성했을 때, 그룹 스케줄링 정책을 지원한다.

즉 철수 와 영희에게 1:1 기회를 준다.
철수 task[0] , 영희 , 철수 task[1] , 엉희 , ..... 철수 task[n] , 엉희

---

## 태스크와 시그널

---

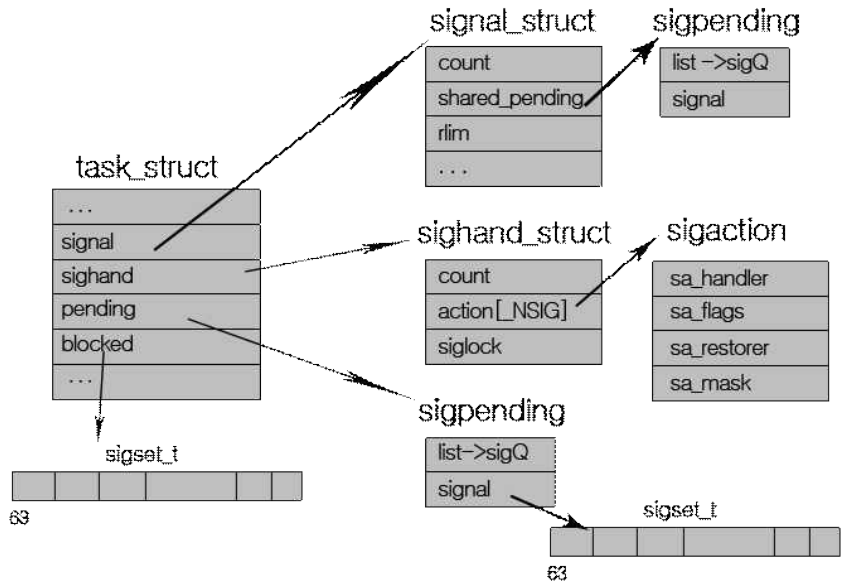
시그널은 태스크에게 비동기적인 사건의 발생을 알리는 매커니즘이다.
-> signal을 통해 정해진 sighandler를 알 수 있었다.

태스크가 시그널을 원할히 처리하려면 다음과 같은 3가지 기능을 지원해야 한다.

첫째. 다른 태스크에게 시그널을 보낼 수 있어야 한다.
-> kill , kill –9 pid , sys_kill(signal,pid)

둘째. 자신에게 시그널이 오면 그 시그널을 수신할 수 있어야 한다.
-> task_struct -> signal, pending 변수 존재.

셋째, 자신에게 시그널이 오면 그 시그널을 처리할 수 있는 함수를 지정할 수 있어야 한다.
-> sys_signal() 이라는게 존재하며 , task_struct 밑에 sighand_struct있음.




시그널을 위해 task_struct에 존재하는 변수들의 기능을 그림으로 나타냈다.
(sig queue 와 sig info 생략)

사용자가 쉘 프롬프트에서 $kill PID 와 같은 명령어를 사용하여 특정 PID를 가지고 있는 태스크를 종료시키려고 한다. 이때 사용자는 PID를 공유하고 있는 쓰레드들이 모두 종료되는 것을 기대할 것이다. 리눅스에서의 PID는 실제로는 tgid를 의미한다. (getpid()를 확인했을 때 tgid를 받아오는 것을 확인했었음) 따라서 PID가 같은 태스크들은 의미상 같은 쓰레드 그룹임을 의미한다. 그러므로 PID를 공유하고 있는 모든 쓰레드들 간에 시그널을 공유하는 메커니즘이 필요하다. 이렇게 여러 태스크들 간에 공유해야 하는 시그널이 도착하게 되면 이를 task_struct 구조체의 signal필드에 저장해 둔다. 이러한 시그널을 보내는 작업은 sys_kill()과같은 시스템 호출을 통해 이뤄진다.

반대로 특정 태스크에게만 시그널을 보내야 하는 경우 task_struct 구조체에 pending 필드에 저장해둔다. signal 이나 pending에 저장할 때는 시그널 번호 등을 구조체로 정의하여 큐에 등록시킨다. 이를 위해 sys_kill()과 같은 시스템 호출을 도입한다.

시그널이 발생했을 때 사용자가 지정한 함수로 설정해주는게 바로 sys_signal()이다. 태스크가 지정한 시그널은 task_struct에 sighand에 저장된다.

특정 시그널을 받지 않도록 설정할수 있는데 이는 task_struct 밑에 blocked를 통해 이뤄진다. 태스크에서 sigprocmask()와 같은 함수를 사용하여 인자로 넘긴 시글널 번호에 해당되는 비트를 블록에서 설정함으로써 특정 시그널을 받지 않게 할수 있다.
하지만 그중 SIGKILL과 SIGSTOP은 블록하거나 무시할수 없다.

수신한 시그널 처리는 태스크가 커널 수준에서 사용자 수준 실행 상태로 전이할 때 (시스템 콜 처리 후 사용자로 넘어올 때) 이루어진다. 커널은 pending의 비트맵이 켜져있는지, 혹은 signal 필드의 count가 0이 아닌지를 검사를 통해 처리 대기중인 시그널이 있는지 확인할 수 있다. 이들 변수가 0이 아니라면 시그널 대기중인지 검사하고, 이 시그널이 블록이 되어 있지 않다면 그 번호에 해당되는 시그널 핸들러를 sighand의 action배열에서 찾아 수행시켜주게 된다.

인터럽트와 트랩 시그널 간의 차이점은 무엇일까?
인터럽트와 트랩이 사건의 발생을 커널에게 알리는 방법이라면, 시그널은 사건의 발생을 태스크에게 알리는 방법이다.

thread_struct -> trapnum -> fault 발생 재실행 pagefault

task_struct -> signal

---

## 4장-1 메모리 관리 기법과 가상 메모리

---

물리 메모리의 한계를 극복하기 위해 여러 가지 기법들이 개발되었는데 그중 하나가 가상메모리 기법이다.

32비트 CPU의 경우 주소 지정할 수 있는 최대 크기인 $2^{32}$크기(4GB)의 가상 주소 공간을 사용자에게 제공하며, 64비트 CPU의 경우 $2^{64}$크기(16GB)의 주소 공간을 사용자에게 제공한다.

32비트 일 때 물리메모리 4GB를 사용자 task에게 전부 제공하는 것이 아니다. 4GB라는 공간은 프로그래머에게 개념적으로 제공되는(가상적으로 제공되는) 공간이며 실제로는 사용자가 필요한 만큼의 물리 메모리를 제공한다.
즉 가상메모리는 사용자에게 개념적으로 4GB를 제공하지만 실제로는 물리 메모리는 필요한 만큼의 메모리만(demand on paging) 사용되므로 가능한 많은 태스크가 동시에 수행될 수 있다는 장점을 제공한다.(컴파일러는 태스크의 배치만 신경씀)

커널의 exec이 물리메모리 배치에 신경쓴다. 즉 운영체제의 sys_exec이 물리메모리에 올려준다.

---

## 물리 메모리 관리 자료구조

---

물리 메모리는 시스템에 없어서는 안 될 귀중한 자원이다.
리눅스는 시스템에 존재하는 전체 물리 메모리에 대한 정보(부팅할 때 뜨는 물리 H/W정보들)를 가지고 있어야 한다.
한정된 용량의 메모리를 효율적으로 사용하기 위해 어떤 정책(paging, buddy, slab, 가상메모리, UMA, NUMA)을 사용하고 있는지 알아보자.

복수개의 CPU를 가지고 있는 컴퓨터 시스템 중 모든 CPU가 메모리와 입출력 버스 등을 공유하는 구조를 SMP(UMA)라 부른다.
복수개의 CPU가 메모리 등의 자원을 공유하기 때문에 성능상 병목 현상이 발생할 수 있다.(메모리가 뜨거워지고 느려지는 현상)

따라서 CPU들을 그룹으로 나누고 그룹에게 별도의 지역메모리를 주는 구조가

생겨났고 이를 NUMA라 한다.

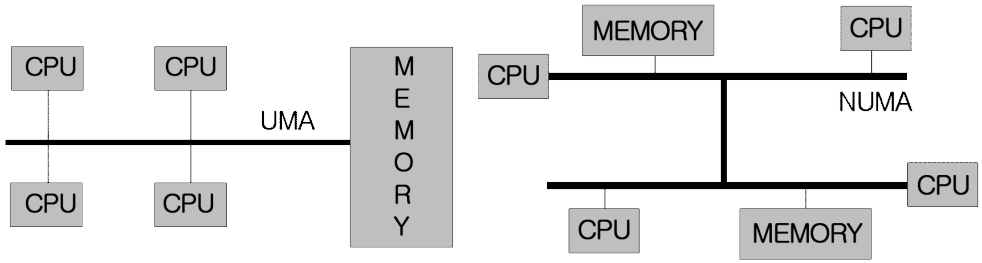
NUMA구조에서는 CPU에서 어떤 메모리에 접근 하느냐에 따라 성능의 차이가 생길수 있다.(CPU의 종류와 속도가 다르기 때문)

---

## NODE

---

리눅스에서 접근 속도가 같은 메모리에 집합을 뱅그(bank)라 부르고 커널에서는 이것을 NODE로 칭한다. UMA구조에서는 1개의 bank가 존재하고 NUMA구조라면 N개의 bank가 존재한다.
이 노드는 리눅스 커널에서 pg_data_t를통해 관리가 되며 NUMA는 복수개의 노드를 리스트형으로 관리해야하기 때문에 pg_data_t(*pgdat) 인 배열형태로 관리한다.




UMA 구조라서 단일 노드를 contig_page_data 변수가 가리키고 있던, 아니면 NUMA 구조라서 pg_data_t(*pgdat)라는 자료구조가 가리키고 있던지 간에, 어쨌든 하나의 노드는 pg_data_t를 통해 표현된다.

이 구조체 구조
node_present_page -> 노드에 속해있는 물리 메모리의 실제 양(페이지 프레임 개수)
node_start_pfn -> 해당 물리메모리가 메모리 맵의 몇 번지에 위치하고 있는지를 나타내는 변수(메모리맵에 몇 번째에 있는지)
node_zones-> zone의 구조체를 담기위한 배열
nr_zones -> zone의 개수를 담는 변수

---

## ZONE

---

ZONE -> 메모리의 특정한 영역 (영역을 분할)

node의 일부분을 따로 관리할 수 있도록 자료구조를 만들어 놓은게 바로 ZONE이다. zone은 동일한 속성을 가지며, 다른 zone의 메모리와는 별도로 관리되어야 하는 메모리의 집합니다.

리눅스의 가상 주소공간과 물리메모리 공간을 1:1로 맵핑하면 아무리 메모리를 사용하려 해봐야 1GB이상은 접근할 수가 없다. 수GB씩의 메모리를 장착하는 현재시스템에서 너무 치명적인 약점이다. 이런 점을 개선하여 물리메모리가 1GB 이상이라면 896MB까지를 커널의 가상주소공간과 1:1 맵핑을 해주고 나머지 부분은 필요할 때 동적으로 연결하여 사용하는 구조를 체택하였다. 필요할 때 동적으로 연결하여 사용하는 구조를 ZONE_HIGHMEM이라 부른다.
즉 896MB과 초과하면 간접접근을 해준다.

주의할점.
시스템에서 언제나 DMA, NOMAL, HIGHMEM 세 개의 zone이 존재하는 것은 아니다. 필요 없다면 한 개의 zone만 존재하는 것 도 가능하다. 예를 들어 64MB SDRAM을 장착하고 있는 ARM CPU 시스템이라면 node 한 개(UMA), zone 한 개가 존재하게 된다.(64bit 는 가상메모리가 커서 HIGHMEM이 없다)

zone 구조체 속에는 zone이 속해있는 물리 메모리의 시작 주소와 크기, free_area 구조체를 담는 변수등이 존재한다.
watermask와 vm_stat를 통해 남아있는 빈 공간이 부족한 경우 적절한 메모리 헤제정책을 결정하게 된다. 또한 프로세스가 zone에 메모리 할당 요철을 하였으나, free페이지가 부족하여 할당해주지 못한 경우 이러한 프로세스들을 wait_queue에 넣고 이를 해싱하여 wait_table변수가 가리키게 한다.
free_area->사용중인 페이지 비어있는 페이지 부분적채워진 페이지가 얼마나

되는지 확인하는 변수

---

## Page frame

---

물리메모리의 최소단위를 page frame이라고 한다.
페이지프레임은 page구조체에 의해 관리된다.
리눅스는 시스템 내의 모든 물리메모리에 접근이 가능해야한다.
이를위해 모든 페이지 프레임 당 하나씩 page 구조체가 존대한다.

*여기서 page는 page frame 보다 작아야되는 것을 필히 알아야한다.
page는 page frame 안에 들어있다.

페이지는 시스템이 부팅되는 순간에 구축되어 물리메모리 특정위치 node_mem_map에서 접근할 수 있다.

복수개의 페이지 프레임은 zone을 구성하며, 때에 따라 하나 혹은 그 이상의 zone이 node를 구성하고 시스템 구조에 따라 하나 혹은 그 이상의 node가 존재하는 것이 리눅스의 전체 물리 메모리 관리 구조이다.

ex) arm은 노드 1개 존 1개

zone -> 노드를 관리한다.
node 가 표현하는 것 -> bank
어러개 있으면 NUMA , 한 개있으면 UMA

---

## Buddy 와 Slab

---

리눅스가 자신이 가지고있는 물리메모리를 할당하고 해제할 때 사용한다.

메모리를 할당할 때 메모리의 최소단위인 페이지프레임(4KB)로 할당하기로 결정하였다.

1. 만약 페이지프레임보다 작은 크기를 요청했을 경우.

30byte를 요청했을 경우 4096byte를 할당해주면 4066byte가 너무 아깝다. 그 결과 내부단편화 문제가 발생한다. 이를 해결하기 위해 slab할당자(slab Allocator)를 도입하였다.

2. 4KB보다 큰 공간을 요청했을 경우.
예를들어 10KB를 요청했을 경우 세 개의 페이지 프레임을 할당하면 내부 단편화를 최소화 시킬 수 있도록 할당 할수 있다 하지만!! 리눅스는 이 요청에 대해 16KB를 할당해주는 버디 할당자를 사용한다. 버디 할당자가 메모리 관리의 부하가 적으며 외부 단편화를 줄일 수 있다는 장점을 제공하기 때문이다.

---

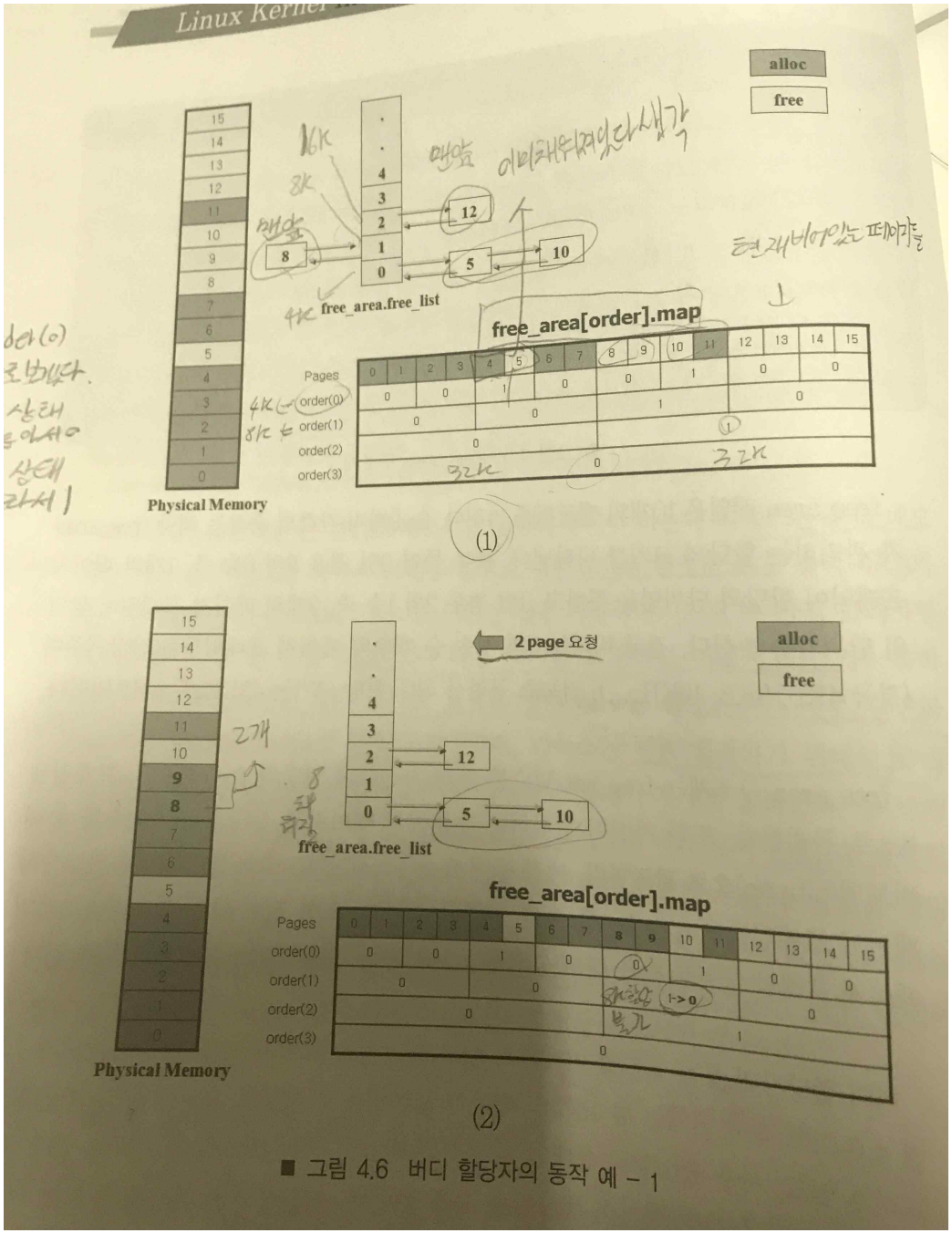
## Buddy Allocator

---

버디 할당자는 zone 구조체에 존재하는 free_area[]배열을 통해 구축된다.
free_area 구조체는 free_list 와 nr_free라는 필드를 갖는다.

free_area는 0~9까지 10개의 index를 가지며 그것은 2의 승수를 나타내고, 2의 승수로 페이지 프레임을 할당의 단위임을 알 수 있다.
(4KB, 8KB, 16KB,..... 4KB x 2^9) 최대 할당 크기는 4MB이다.

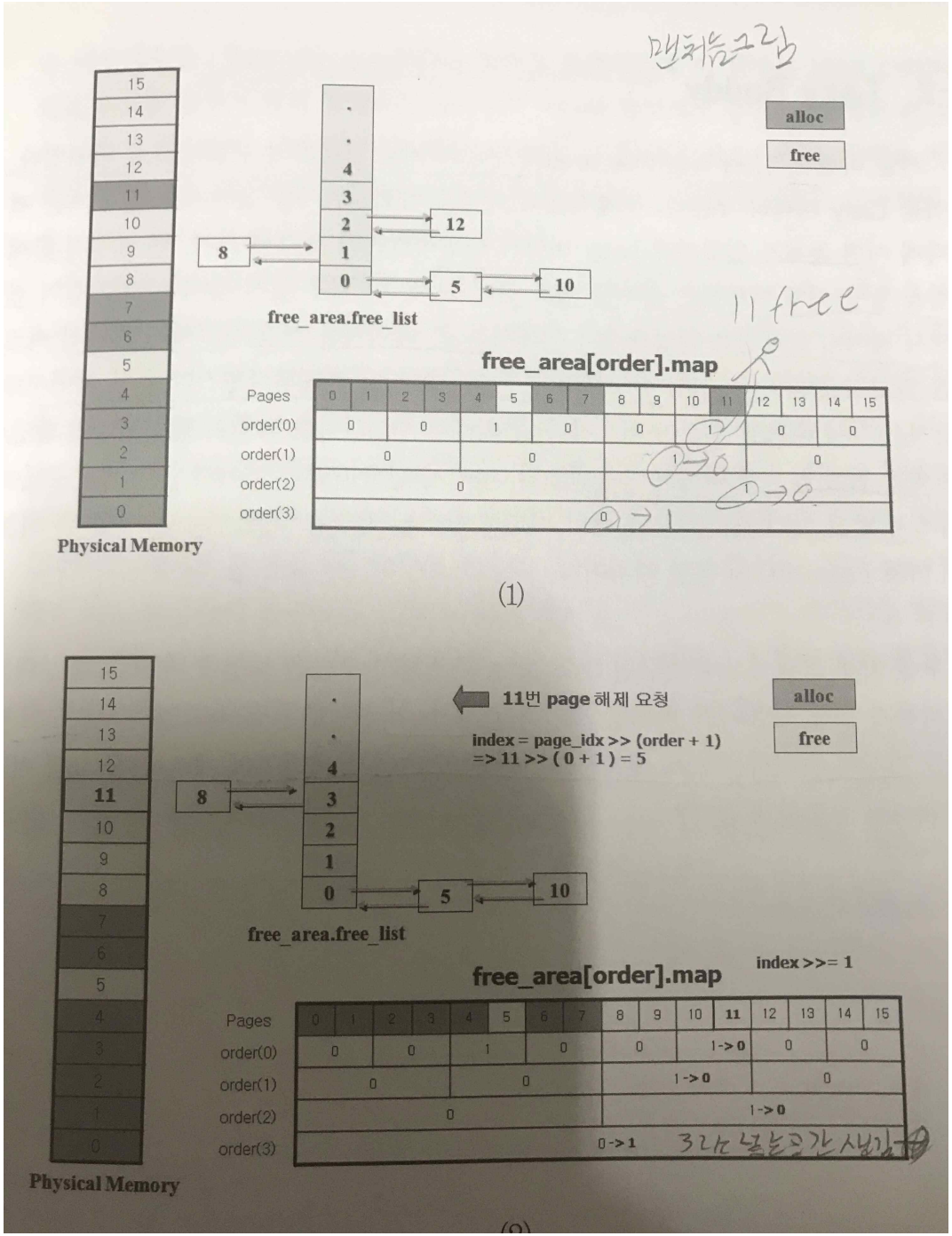
free_area 구조체는 free_list 변수를 통해 자신에게 할당된 free 페이지 프레임을 list로 관리한다. 또 nr_free를 통해 비트맵으로 관리한다. 예를 들어 free_area[1]에는 free 상태인 연속된 2개의 페이지 프레임이 list를 통해 연결되 있다.




■ 그림 4.6 버디 할당자의 동작 예 – 1

왼쪽 그림을 보면 2개의 page가 요청이 되었다.
여기서 order 는 free_area라고 봐도 될거같다.
order(0)은 2^0을 의미한다.
즉 4KB를 페이지 프레임 하나를 의미하고
list로 통해 관리되어 연속되어 연결되어있으므로 0번과 1번
2번과 3번 ..... 14번과 15번을 비교할수 있다.
두 개를 비교하요 XOR연산을 해준다.
order(1)은 2^1 이므로 8KB를 의미한다
즉 페이지프레임 2개를 한묶음으로 보고 비교하라는 뜻이다.
order(2) 는 16KB
order(3) 은 32KB 로 비교한다.

페이지 프레인 한 개라도 채워져있으면 데이터가 있는걸로 인식되고
아무것도 없을 경우 데이터가 없는 free 상태로 인식된다.




왼쪽의 그림은 해제할때의 그림이다. 해제할때도 요청할때와 동일한 방식으로 비교한다. 다만 주의해야할 점이 있다면 1번 그림에 12가 사라진 것을 볼 수 있다. 이는 11번이 해제되면서 free상태인 프레임이 32KB가 되었고 그 뜻은 order(3)을 의미하며 32KB시작은 8 이기 때문에 8이 1번에서 3번으로 옮겨가고 2번에 셋팅되어있던 12가 사라진 형태의 모습을 볼 수 있다.

---

## Lasy Buddy

---

buddy 의 free 의 방법이 바뀐 방법이다.

한 페이지 프레임을 할당/해제 반복하는 경우 페이지 프레임을 할당해주기 위해서는 큰 페이지를 쪼개서 할당해주고 또 해제하면 그걸 다시 큰페이지로 합쳐서 관리했다. 이런 작업이 반복되면 할당/해제 때문에 많은 오버헤드가 동반된다.(속다가 매우 느려짐) 따라서 할당된 페이지 프레임을 합치지 말고, 곧 다시 할당 될 테니 합치는 작업을 뒤로 미룬다.

nr_free => 자신이 관리하는 zone내에서 비사용중인 페이지 프레임의 개수이다.
buddy는 zone마다 유지되고 있는 watermark(메모리가 얼마나 차있는지 관리, 만약 메모리가 부족하면 데이터를 해제하라 말해줌) 값과 현재 사용가능한 페이지 수를 비교한다. 이를 통해 zone에 가용메모리가 충분한 경우 해제된 페이지를 병합 작업을 최대한 뒤로 미룬다. 만약 가용메모리가 부족해 지는 경우 다음과 같이 병합 작업을 수행한다.

1. 버디 메모리를 반납하는 함수인 __free_pages()함수가 내부적으로 __free_one_page()함수를 호출한다. 이함수는 MAX_ORDER만큼 루프를 돌면서 현재 해제하는 페이지가 버디와 합쳐져서 상위 order에서 관리될 수 있는지 확인한다.(앞에서 한과정) 가능한 경우 현재 order의 nr_free를 감소시키고, 상위로 페이지를 이동시킨 뒤 , 상위 order는 nr_free를 증가시킨다.

---

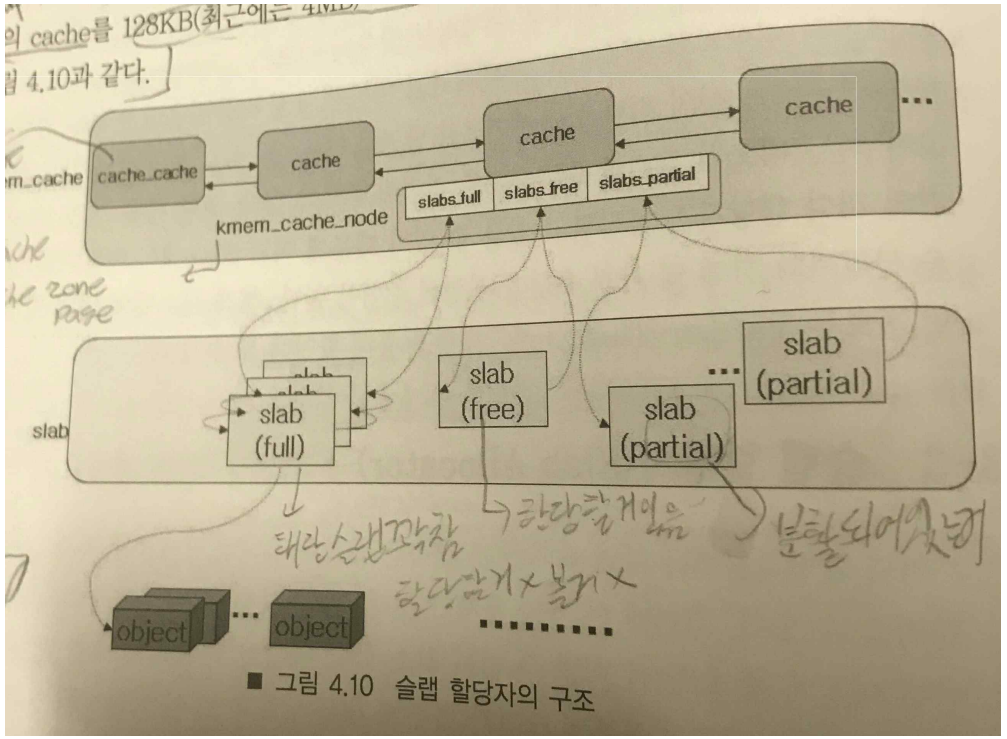
## Slab Allocator

---

slab은 32byte공간이 128개 있다고 생각하고 시작해야한다. 32바이트를 요청하면 버디 할당자로부터 할당받아오는 것이 아니라 미리 할당받아 분할하여 관리하고 있던 바로 이 공간에서 떼어 주는 것이다. 해제한다면 역시 버디로 반납하는 것이 아니라 미리 할당 받아 관리하던 공간에서 다시 가지고 있으면 된다. 마치 일종의 캐시로 사용하는 것이다.

이러한 캐시 집합으로 메모리를 관리하는 정책을 바로 slab할당자라 부른다.

캐시는 어떤 크기를 가지고 있어야할까?
자주 할당되고 해제되는 크기의 캐시를 가지고 있어야 내부 단편화를 최소화시킬 수 있다. 따라서 태스크가 생성되고 제거될 때마다 할당/해제되는 task_struct를 위한 공간처럼 커널 내부에서 자주 할당/해제 되는 자료구조의 크기를 위한 캐시를 유지한다.
일반적으로 32바이트에서부터 시작되는 2의 승수 크기의 캐시를 128KB까지 (최근에는 4MB)까지 유지한다.



■ 그림 4.10 슬랩 할당자의 구조

슬랩할당자는 외부 인터페이스 함수로 kmalloc()/kfree()를 제공한다.
(커널도 프로그램이다 메모리가 있어야한다. 그때 쓰는게 kmalloc)
kmalloc은 메모리를 순차적으로 배치한다.