1. 메모리 구조

(1) Virtual memory (가상 메모리):

가상 메모리는 실제 시스템에 존재하는 물리 메모리의 크기와 관계없이 가상적인 주소 공간을 사용자 태스크에게 제공한다. (32bit의 경우 2의 32승이 최대 크기이기 때문에 4GB) -> 가상적으로 줄거라면 기왕이면 넓은 공간을 주는 것이 좋기 때문. 한가지 주의할 점은 물리적으로 4GB의 메모리를 모두 제공하는 것은 아니라는 것이다.
결국 가상 메모리는 사용자에게 개념적으로 큰 공간을 제공함과 동시에 물리 메모리는 필요한 만큼의 메모리만 사용되므로 가능한 많은 태스크가 동시에 수행될 수 있다는 장점을 제공한다.

가상 메모리의 주요한 기능은 다음 3가지로 요약할 수 있다.

- **주 기억장치의 효율적 관리** : 주 기억장치를 하드디스크에 대한 캐시로 설정하여, 당장 사용하는 영역만 유지하고 쓰지 않는 데이터는 하드디스크로 옮긴 뒤, 필요할 때만 램에 데이터를 불러와 올리고 다시 사용하지 않으면 하드디스크로 내림으로써 램을 효과적으로 관리한다.

- **메모리 관리의 단순화** : 각 프로세스 마다 가상메모리의 통일된 주소 공간을 배정할 수 있으므로 메모리 관리가 단순해진다.

- **메모리 용량 및 안정성 보장** : 한정된 공간의 램이 아닌 거의 무한한 가상메모리 공간을 배정함으로써 프로세스들끼리 메모리 침범이 일어날 여지를 크게 줄인다.

(2) 물리 메모리

- **SMP** (symmetric multiprocessing)

두 개 또는 그 이상의 프로세서가 한 개의 공유된 메모리를 사용하는 다중 프로세서 컴퓨터 아키텍처이다. 현재 사용되는 대부분의 다중 프로세서 시스템은 SMP 아키텍처를 따르고 있다. UMA(Unified Memory Access)라고도 불린다.

- **NUMA**(Non-Uniform Memory Access)

메모리에 접근하는 시간이 CPU와 메모리의 상대적인 위치에 따라 달라지는 컴퓨터 메모리 설계 방법. 각 CPU는 메모리의 일부를 자신의 지역 메모리(Local Memory)로 가지고 있으며 이 지역 메모리에 접근하는 속도는 원격 메모리(Remote Memory)에 접근하는 속도보다 훨씬 빠르다.
간단히 예를 들어 시스템에 CPU 소켓이 네 개 있고, 512GB의 메모리가 설치되어 있다면 물리 주소 0~128G-1번지까지의 메모리는 0번 소켓의, 128G~256G-1번지까지의 메모리는 1번 소켓의 지역 메모리가 되는 방식. 각 소켓에서 지역 메모리에 접근할 때에는 원격 메모리에 접근할 때보다 훨씬 빠르게 데이터를 읽고 쓸 수 있다. 하나의 CPU 소켓에 코어 여러개가 들어가 있을 수 있기에 같은 지역 메모리를 사용하는 CPU 코어들을 묶어서 하나의 NUMA 노드로 친다. 8코어 4소켓 CPU라면 (하이퍼스레딩을 가정하지 않을 때에) 0~7번 코어는 NUMA 노드 0번, 8~15번 코어는 NUMA 노드 1번과 같은 방식.

- Node

리눅스에서는 접근 속도가 같은 메모리의 집합을 bank라고 한다. 이 bank를 표현하는 구조를 Node라고 하는데 하나의 Node는 pg_data_t 구조체로 표현된다. (만약 UMA구조에서 리눅스가 수행된다면 한개의 node가 존재할 것이다.)

- Zone

노드 안의 존재하고 있는 메모리는 모두 어떠한 용도로 사용될 수 있어야 한다. 노드에 존재하는 물리 메모리 중 16MB이하 부분을 좀 특별하게 관리해 놓았는데 이 자료구조를 zone 이라고 부른다.
다시 말하면 zone은 동일한 속성을 가지며, 다른 zone의 메모리와는 별도로 관리되어야 하는 메모리의 집합이다. 리눅스에서 16MB 이하의 메모리는 ZONE_DMA 라 하고, 이상의 메모리를 ZONE_NORMAL이라고 한다.
32bit system에서는 리눅스 커널은 1GB의 가상 공간을 차지한다. 따라서 물리 메모리가 1GB가 넘어간다면 1대 1로 매핑할 수 없다. 따라서 물리 메모리가 1GB 이상이라면 896MB(아키텍쳐마다 다르다.)는 물리 메모리와 1대 1로 매핑해주고 그 이상은 필요할 때 동적으로 물리 메모리에 할당해준다. 1대 1로 매핑되는 부분을 ZONE_NORMAL이라 하고, 그 이상의 부분을 ZONE_HIGHMEM이라 한다. 1대 1로 물리 메모리와 매핑되는 ZONE_DMA, ZONE_NORMAL은 커널 메모리 할당 시 가장 빠르게 접근할 수 있는 영역이다. 또한 32bit에서는 영역의 크기가 제한되어 있기 때문에 cost가 비싼 영역이고 물리 메모리가 할당될 때, ZONE_HIGHMEM이 모두 소진된 후에 할당이 된다.
한 가지 주의할 점은 모든 시스템에서 언제나 DMA, NORMAL, HIGHMEM 세 개 의 zone이 존재하는 것은 아니라는 것이다. 예를들어 64MB의 ARM CPU 시스템이라면 node 한 개 zone 한 개가 존재하게 된다.

- Page Frame

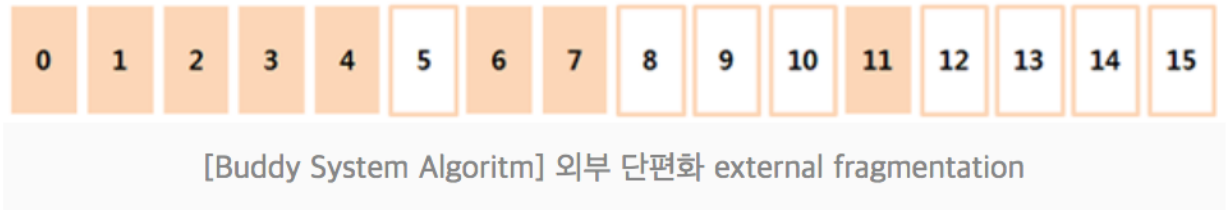
물리 메모리의 최소 할당 단위를 말한다. (가상 메모리의 최소 할당 단위는 페이지)
즉 여러 개의 페이지 프레임이 zone을 구성하고 하나 이상의 zone이 node를 구성하고, 하나 이상의 node가 리눅스 전체 물리 메모리를 관리한다.

2. Buddy와 Slab

리눅스는 자신이 가지고 있는 물리 메모리를 어떻게 할당 또는 헤제할까? 리눅스는 물리 메모리의 최소 관리 단위인 page frame 단위로 할당을 한다. 결국 4KB가 최소 할당단위가 된다. (8KB, 2MB 등 크기는 설정 가능하다)

- Buddy Allocator

최소 단위인 4KB보다 큰 공간을 요청하면 어떻게 될까? 이 경우 세 개의 페이지 프레임을 할당하면 내부 단편화를 최소화 할 수 있다. 하지만 리눅스는 이 요청에 대해 16KB를 할당해주는 버디 할당자를 사용한다. 버디 할당자가 메모리 관리의 부하가 적으며 외부 단편화를 줄일 수 있다.
Buddy 할당자는 ZONE 구조체에서 free_area[] 배열에 의해서 구축된다. free_area는 10개의 엔트리를 가지고 있는데 free_area가 관리하는 할당의 크기를 나타내며, 즉, 4MB(2^10 * 4KB)까지 할당할 수 있다.
Free_area 구조체는 free_list와 map을 통해 할당된 페이지 프레임을 list와 bitmap으로 관리한다.



[Buddy System Algoritm] 외부 단편화 external fragmentation

예를 들어 색으로 칠해진 페이지 프레임들을 사용하고 있고 나머진 free 상태라 볼 때, 색으로 칠해진 비트가 할당되고, 색으로 칠해진 비트가 해제되었다고 하면,

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

Order(0)는 free_area[0]와 같은데, order(n)은 연속된 2^n개의 페이지 프레임이 free 상태로 존재하는지를 나타내 준다. 예를 들어 2개의 연속된 페이지 프레임을 할당받아야 할 때, 비트맵에서 order(1) 부분을 확인해 1로 된 부분이 있으면 bitmap을 0으로 수정하고 할당해 준다. 만약 없다면, 보다 큰 order(n)에서 연속된 2^n개의 페이지 프레임을 분할해, 할당하고 bitmap을 수정해준다. 이때, 나누어진 두 부분을 buddy라고 부르며 이때문에 buddy allocator라고 부른다. 해제할때도 마찬가지로, 해당 부분을 해제하고 bitmap을 수정해준다.

결국 버디 할당자는 요청된 크기를 만족하는 최소의 order에서 페이지 프레임을 할당해 준다. 만일 그 order에 가용한 페이지 프레임이 존재하지 않으면 상위 order에서 페이지 프레임을 할당받아 두 부분으로 나누어, 한 부분은 할당해주고 나머지 부부은 하위 order에서 가용 페이지로 관리한다.

- Lazy Buddy

커널 버전 2.6.19 이후 등장한 Lazy buddy는 할당과 해제로 인한 오버헤드를 줄이기 위한 할당자 구조다. Buddy는 ZONE마다 유지되고 있는 watermark의 값과 사용가능한 페이지 프레임 수를 비교한다. 가용 메모리가 충분한 경우 페이지 병합 작업을 최대한 미루고 메모리가 부족해질 때 병합 작업을 수행한다. 메모리를 반납하는 함수는 __free_pages()이고, 할당하는 함수는 __alloc_pages()다. __free_pages() 함수는 내부적으로, MAX_ORDER(위에서는 10)만큼 loop를 돌며, 해제된 해당 페이지 프레임이 버디(옆 메모리)와 합쳐져 상위 order에서 관리될 수 있는지 확인하고, 가능하다면 order의 nr_free를 변경해준다.

- Slab Allocator

커널이 관리하는 페이지 단위는 커널에서 사용하기에 다소 큰단위이다. 4K가 응용 프로그램을 개발하는 분들은 쉽게 생각할지 모르겠지만 커널의 입장에서 많은 함수들이 4K씩 사용한다는 것은 시스템의 치명적인 성능저하를 유발할 수도 있으며 메모리 단편화를 너무 많이 일으켜 금방 시스템의 성능저하로 이어지게 될 것이다. 그러므로 커널 입장에서는 몇 byte요청을 4K로 되돌려 준다는 것은 용납할 수 없는 일이된다.
예를 들어19byte를 요청한 입장에서 4K가 할당되었다는 것은 나머지 4077 byte는 사용자가 free하지 않는 한 시스템의 입장에서는 사용하지 못하는 영역이 되버리고 말기 때문이다.
이를 우리는 멋있는 말로 내부 단편화라고 한다. 내부 단편화문제를 해결하기 위하여 슬랩 할당자가 나오게 되었다.

따라서 자주 할당되고 헤제되는 크기의 cache를 가지고 있어야 내부 단편화를 최소화 시킬 수 있다. 커널 내부에서 자주 할당/해제 되는 자로구조의 크기를 위한 cache를 유지해야 한다.

3. 가상 메모리 관리 기법

태스크는 자신의 고유한 가상 메모리를 갖는다. 따라서 커널은 태스크의 가상 메모리가 어디에 존재하는지를 관리해야 한다. 즉 어디에 text 영역이 있고 어디에 data 영역이 있는지, 그리고 어느 영역이 사용 중이며 어느 영역이 사용 가능한 지의 정보를 알고 있어야 한다.
태스크는 텍스트, 데이터, 스택 등의 region으로 구성된다. 텍스트, 데이터, 힙영역은 가상 메모리의 0번지부터 차근차근 올라가며, 스택영역은 커널과 사용자 영역의 경계인 3GB바로 아랫부분에 존재한다.