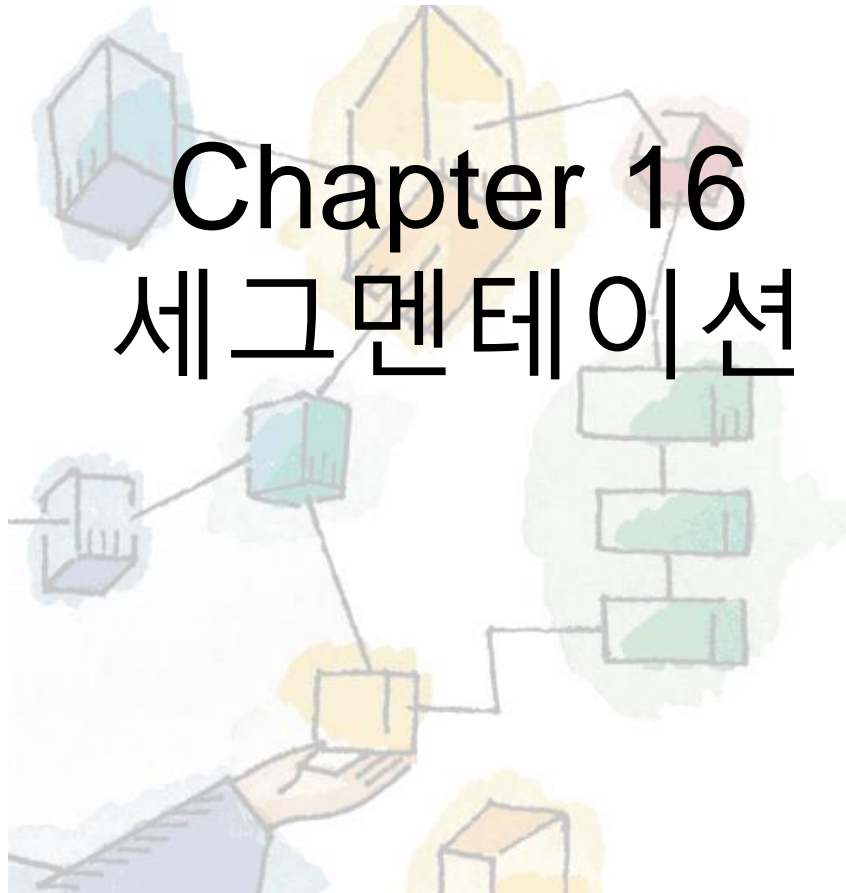


Chapter-10

운영 체제

정내훈

2023년 가을학기
게임공학과
한국공학대학교

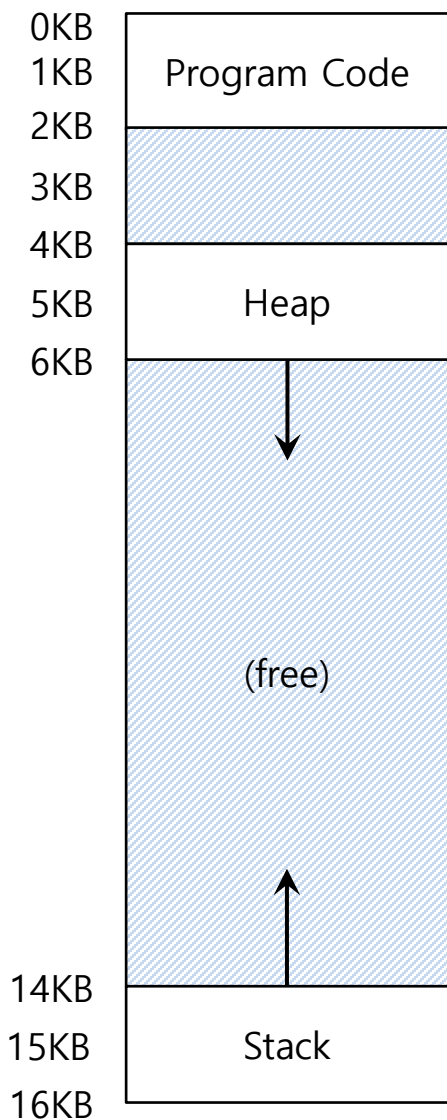
A hand-drawn illustration of a city with various buildings and a hand holding a yellow cube. The scene includes a blue cube, a yellow cube, a green cube, and a hand holding a yellow cube. The background is a light blue sky with a yellow sun. The text "Chapter 16" and "세그멘테이션" is overlaid on the image.

Chapter 16

세그멘테이션



베이스와 바운드방식의 비효율성



- 너무 큰 “빈” 영역
- “빈” 공간이 물리메모리를 차지한다.
- 맞는 주소 공간이 물리메모리에 없으면 실행이 어렵다.
 - 너무 크거나, 쪼개져 있거나...

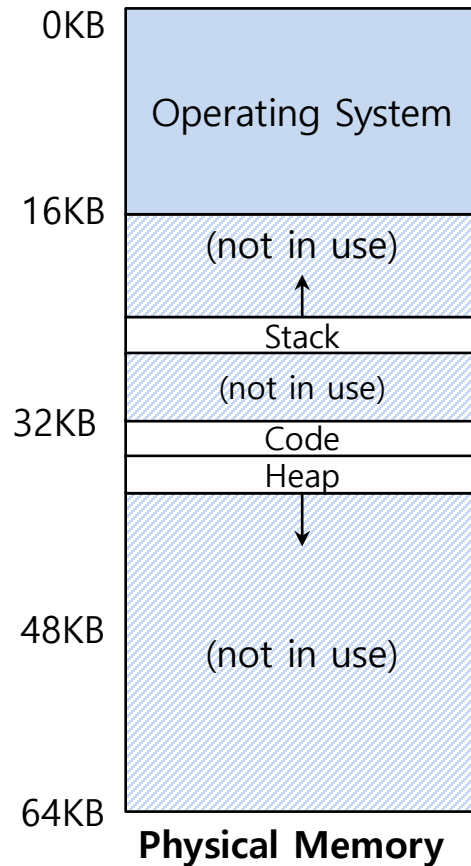


세그멘테이션

- 세그먼트는 일정한 크기의 연속된 메모리 공간을 뜻한다.
 - 여러 종류의 세그먼트가 있다 : code, stack, heap
 - 논리적으로 하나의 프로세스를 여러 개의 세그먼트로 나눌 수 있다.
- 각 세그먼트는 물리메모리의 **여러 곳에 산재**할 수 있다.
 - 세그먼트 별로 베이스와 바운드가 존재한다.



실제 메모리에서 세그먼트 할당



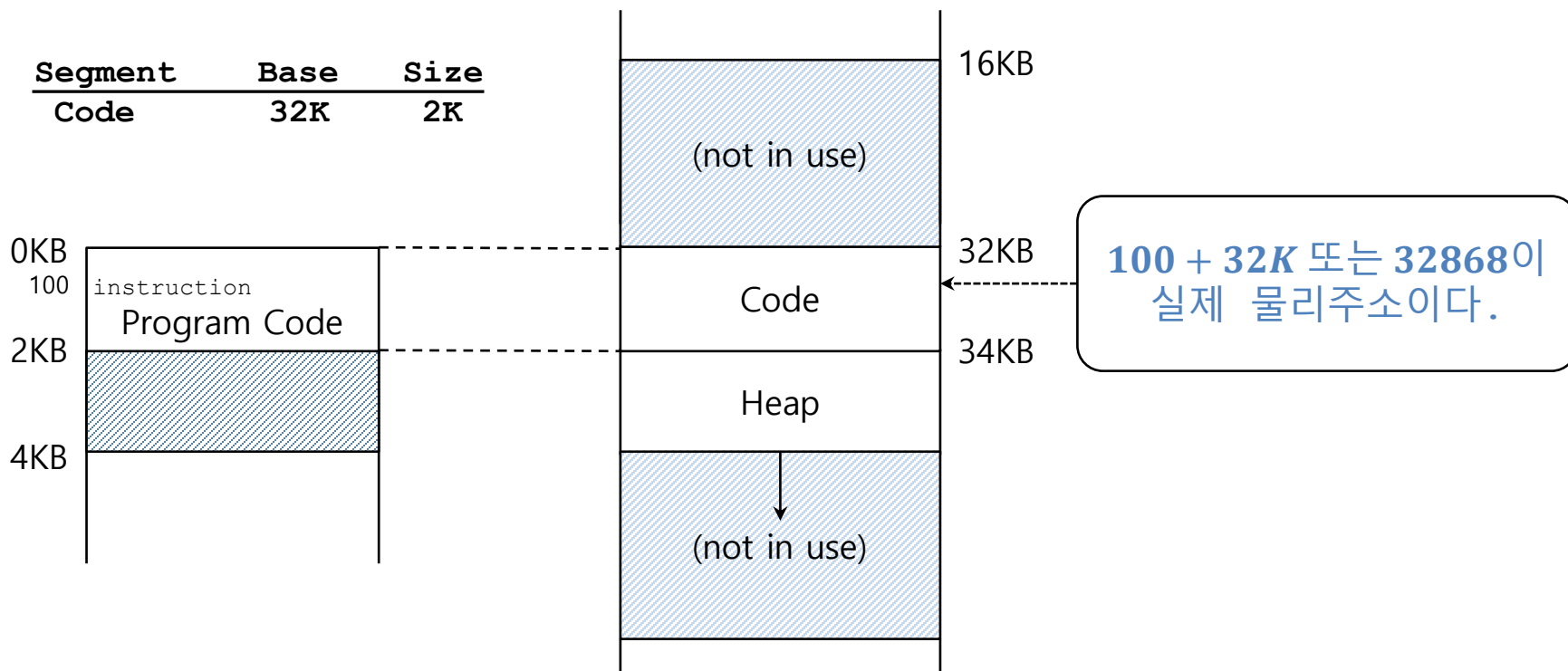
Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K



세그먼트의 주소변환

$$\text{물리주소} = \text{offset} + \text{base}$$

- 가상 주소 100의 offset은 100.
 - 코드 세그먼트의 가상 주소가 0 부터 시작할 때.

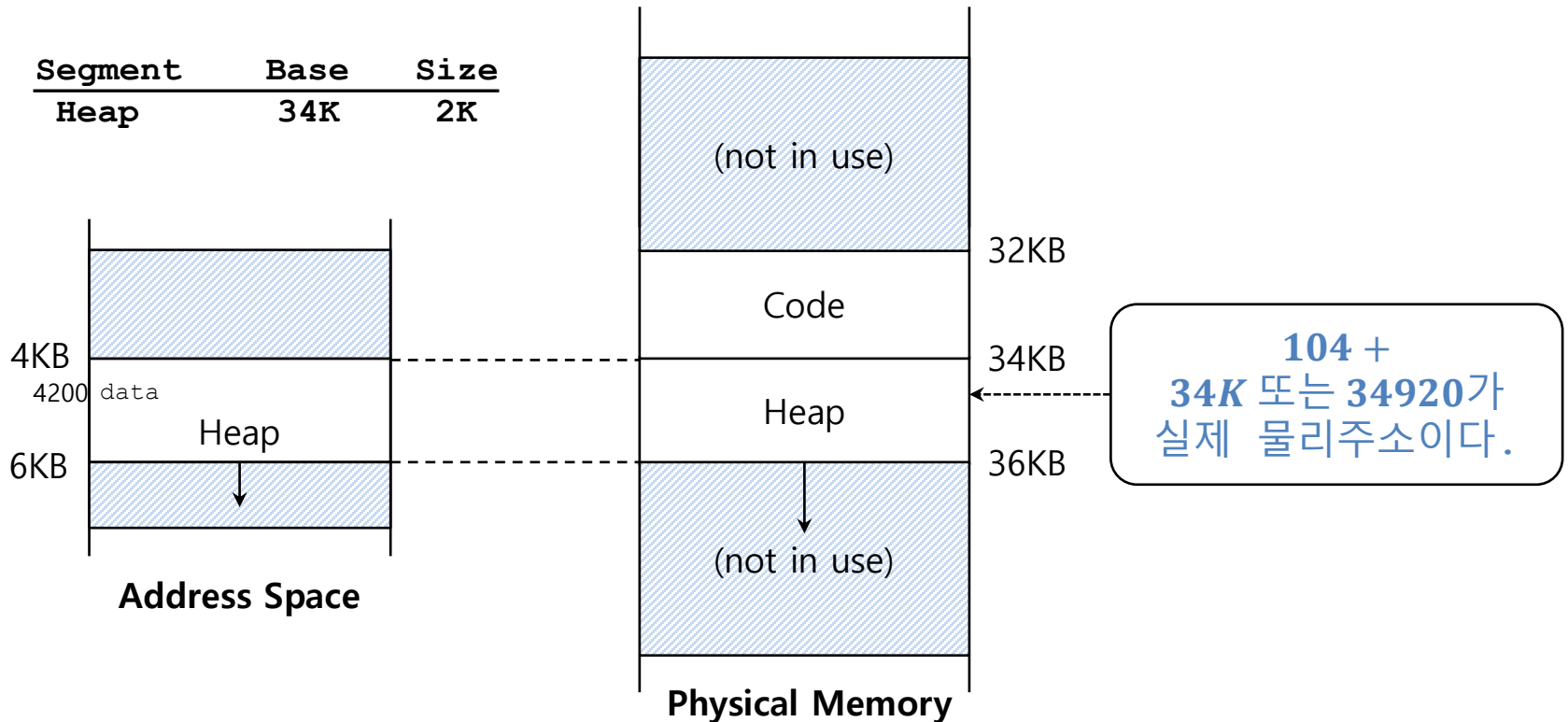




세그먼트의 주소변환 (계속)

가상주소 + *base*가 물리주소가 아니다.

- 가상 주소 4200의 offset 104이다.
 - 힙 세그먼트의 가상 주소 시작은 4096이다.

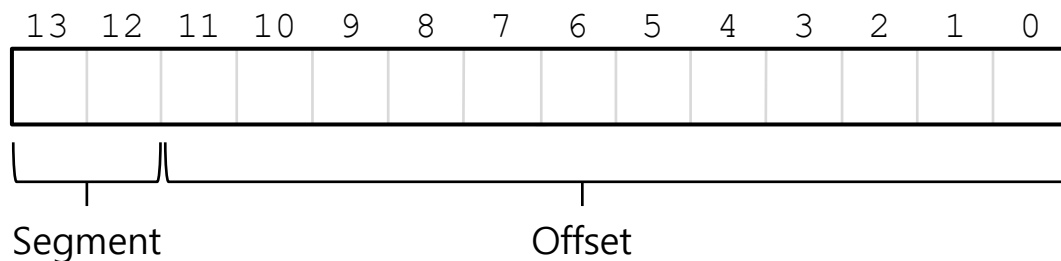




세그먼트 주소

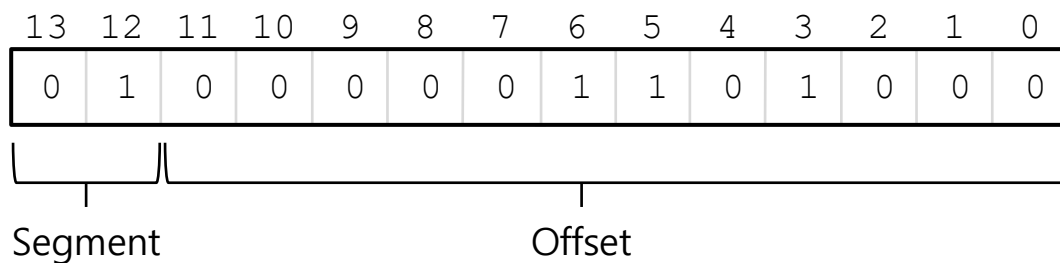
- 명시적 접근(Explicit approach)

- 주소공간의 상위 몇 비트를 세그먼트 지정에 사용



- 예: 가상주소 4200 (010000001101000)

Segment	bits
Code	00
Heap	01
Stack	10
-	11





세그먼트 주소

HW 구현

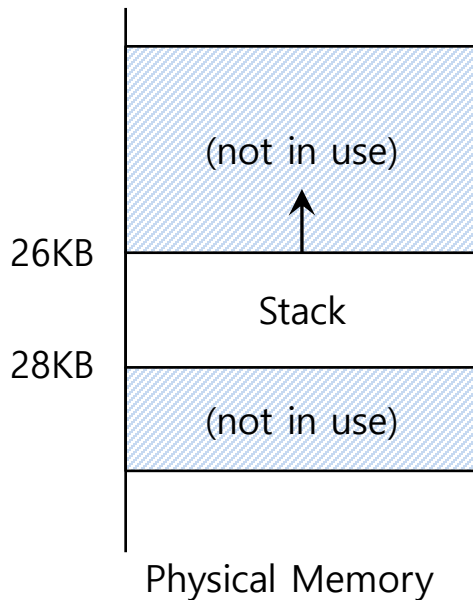
```
1 // get top 2 bits of 14-bit VA
2 Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
3 // now get offset
4 Offset = VirtualAddress & OFFSET_MASK
5 if (Offset >= Bounds[Segment])
6     RaiseException(PROTECTION_FAULT)
7 else
8     PhysAddr = Base[Segment] + Offset
9     Register = AccessMemory(PhysAddr)
```

- SEG_MASK = 0x3000 (1100000000000000)
- SEG_SHIFT = 12
- OFFSET_MASK = 0xFFF
(0011111111111111)



스택 세그먼트 변환

- 스택은 **거꾸로** 자란다.
- **추가 하드웨어 지원**이 필요.
 - 세그먼트의 확장 방향 결정.
 - 1: positive direction, 0: negative direction



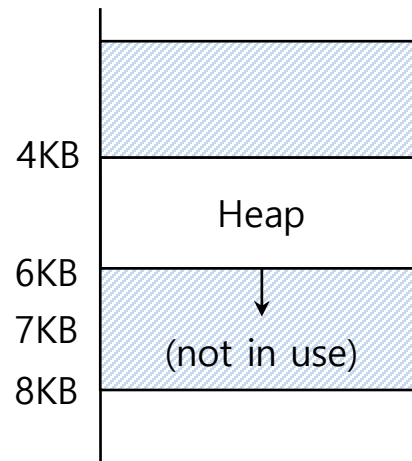
Segment Register(with Negative-Growth Support)

Segment	Base	Size	Grows Positive?
Code	32K	2K	1
Heap	34K	2K	1
Stack	28K	2K	0



세그먼트 오류(Fault, Violation)

- 7KB같은 세그먼트 영역 밖의 잘못된 주소(**illegal address**)에 접근하려 할 때 OS 세그먼트 폴트(segment fault) 또는 세그먼트 위반(segment violation) 처리를 해야 한다.
 - HW가 메모리 접근을 검사해서 오류 시 interrupt를 발생시킨다.



Address Space



공유 지원

- 같은 물리 메모리를 여러 세그먼트에서 공유 할 수 있다. (보통 다른 프로세스)
 - 코드공유(**Code sharing**) 형식으로 많이 사용
 - 보호를 위해 추가 하드웨어 지원이 필요하다.
- 추가 하드웨어 지원은 **Protection bits**가 필요
 - 세그먼트당 **몇 개의 추가 bit**로 읽기, 쓰기, 실행 권한을 표시.
 - C의 const 에서도 사용, 생산자/소비자 프로세스 에서 사용 가능

Segment Register Values(with Protection)

Segment	Base	Size	Grows	Positive?	Protection
Code	32K	2K		1	Read-Execute
Heap	34K	2K		1	Read-Write
Stack	28K	2K		0	Read-Write



대단위 대 소단위 세그멘테이션

- **대단위(Coarse-Grained)**는 적은 개수의 정해진 세그먼트 사용.
 - e.g., code, heap, stack.
- **소단위(Fine-Grained)**는 주소 공간관리에 더 유연하다.
 - 큰 **Segment table** 하드웨어가 필요하다.

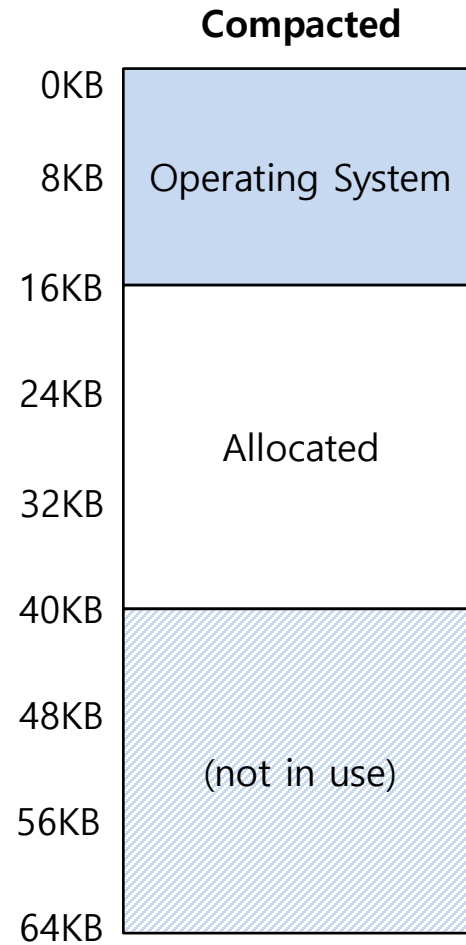
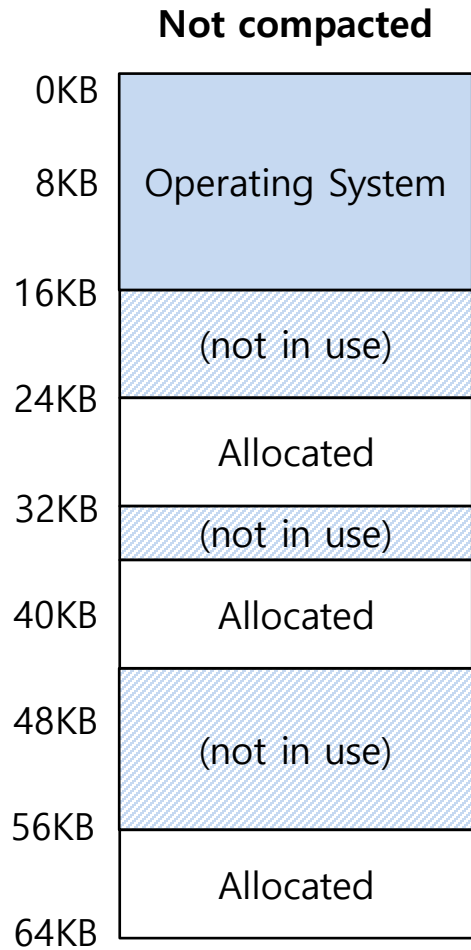


OS 지원: 단편화(Fragmentation)

- **외부 단편화(External Fragmentation):** 작은 크기의 빈 공간들이 많이 생기는 현상. 너무 작아서 사용하기 어려움
 - 전부 24KB가 비어 있지만, 여기저기 흩어져 있음.
 - 20KB의 요청을 OS가 들어줄 수 없음.
- **압축 (Compaction):** 기존의 세그먼트들의 물리적 위치를 재조정(**rearranging**) 하는 것.
 - 압축에는 **비용**이 든다.
 - **Stop** 실행 중인 프로세스.
 - **Copy** 메모리 내용을 다른 곳에.
 - **Change** 세그먼트 레지스터들의 내용.



메모리 압축



17. 빈 공간 관리



빈 공간 관리의 필요성

- 운영체제에서 세그먼트단위로 메모리를 관리할 때
 - HW에서 Paging을 지원하지 않을 때
- C/C++의 라이브러리에서 메모리 할당을 구현할 때
 - new/delete, malloc/free를 sbrk나 mmap으로 구현할 때
- 고성능 프로그램에서 성능향상을 위해 메모리 할당을 직접 구현할 때
 - sbrk나 mmap호출 최소화
 - 멀티쓰레드 메모리 할당, NUMA메모리 할당



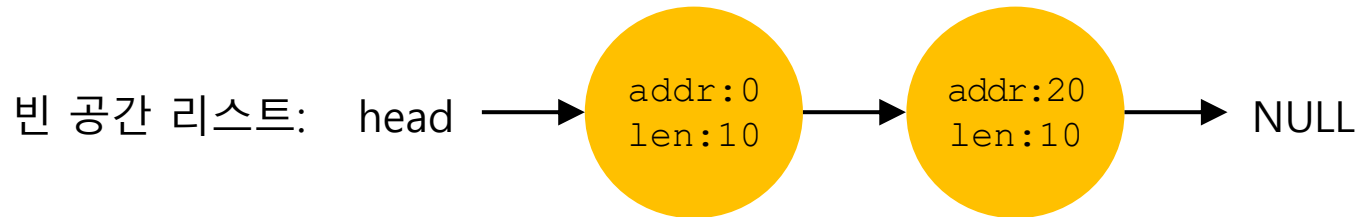
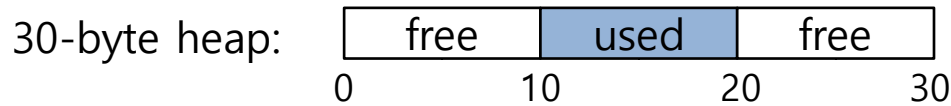
단편화 관리 기법

- 저수준 기법
 - 분할과 병합
 - 개별 리스트
- 버디 시스템



분할

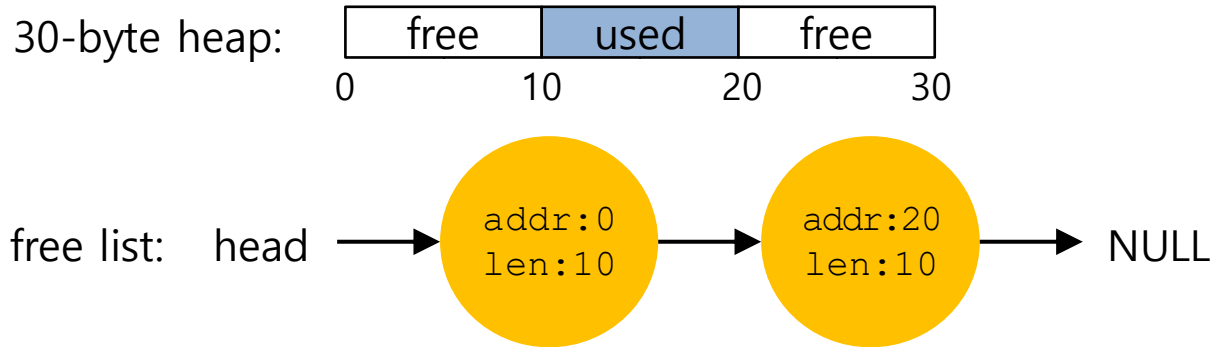
- 메모리 할당은 기존의 빈 공간을 여러개로 쪼갤 수 있다.
 - 빈 공간 보다 필요한 메모리 크기가 **작은** 경우
 - (예) 30byte의 빈 공간이 10byte 요청으로 인해 3개로 쪼개지는 경우



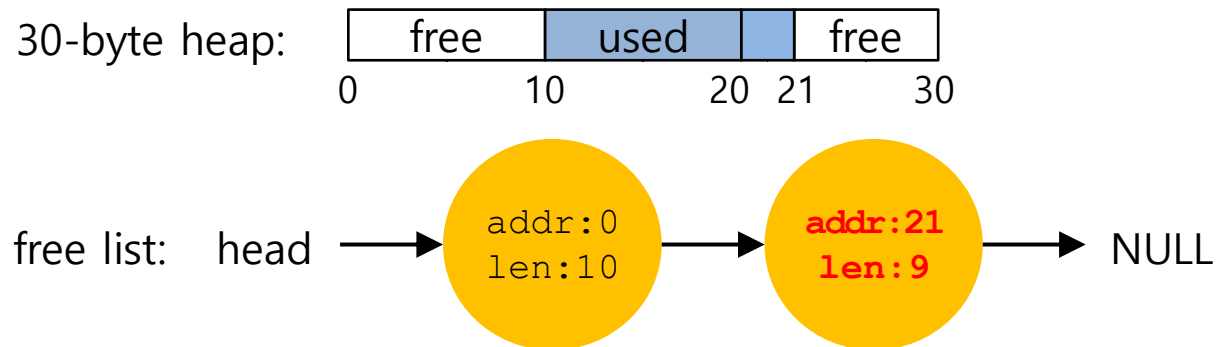


분할(Cont.)

- 1 byte 요청이 있을 때.



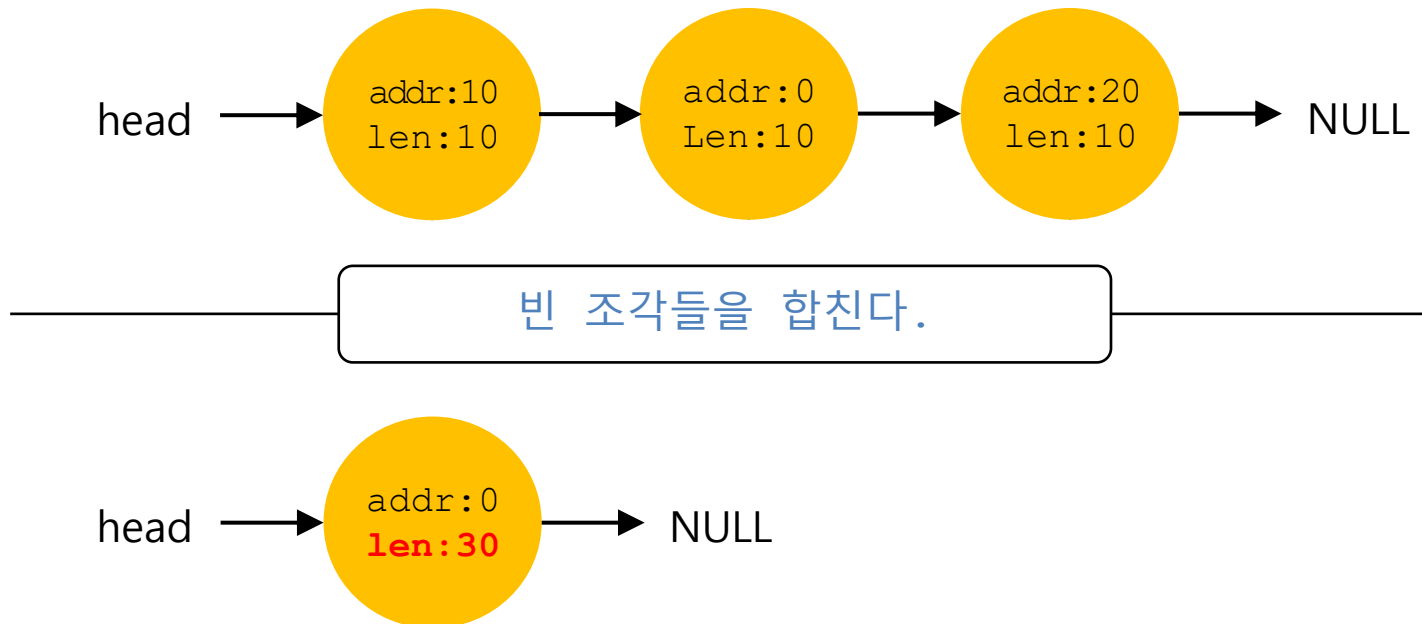
10 바이트 크기의 빈공간을 분할한다.





병합(Coalescing)

- 가장 큰 빈 공간보다 큰 크기의 요청이 있으면 요청을 들어줄 수 없다.
- 병합: 빈 공간이 이웃해 있는 경우 하나로 합친다.

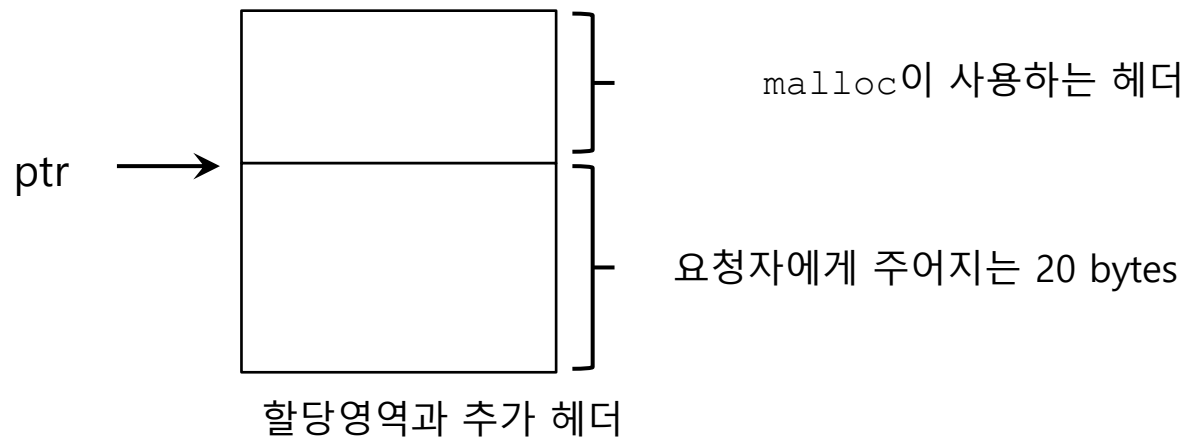




할당영역 크기 관리

- `free(void *ptr)` 는 크기를 매개변수로 받지 않는다.
 - 라이브러리가 어떻게 빈공간 리스트에 넣을 크기를 알 수 있는가?

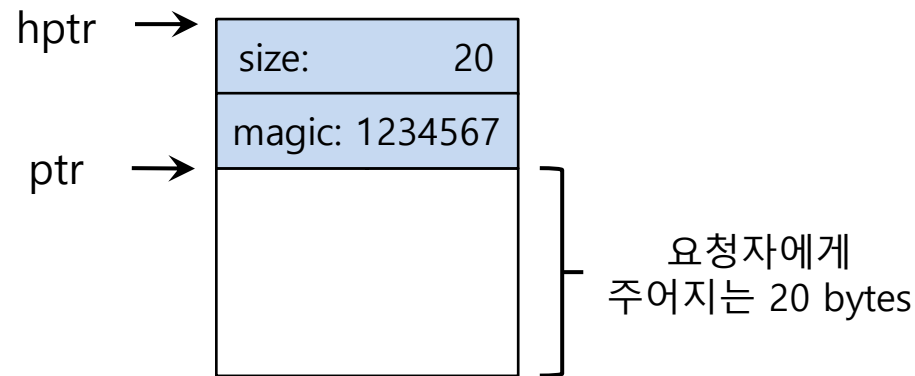
```
ptr = malloc(20);
```





메모리 조각(청크) 헤더

```
typedef struct __header_t {  
    int size;  
    int magic;  
} header_t;
```





메모리 조각(청크) 헤더

WINDOWS의 HEADER

```
typedef struct _CrtMemBlockHeader
{
    // Pointer to the block allocated just before this one:
    struct _CrtMemBlockHeader *pBlockHeaderNext;
    // Pointer to the block allocated just after this one:
    struct _CrtMemBlockHeader *pBlockHeaderPrev;
    char *szFileName;        // File name
    int nLine;               // Line number
    size_t nDataSize;        // Size of user block
    int nBlockUse;           // Type of block
    long lRequest;           // Allocation number
    // Buffer just before (lower than) the user's memory:
    unsigned char gap[nNoMansLandSize];
} _CrtMemBlockHeader;

/* In an actual memory block in the debug heap,
 * this structure is followed by:
 *   unsigned char data[nDataSize];
 *   unsigned char anotherGap[nNoMansLandSize];
 */
```




메모리 조각 헤더

```
void free(void *ptr) {  
    header_t *hptr = (void *)ptr - sizeof(header_t) ;  
    ...  
    assert(hptr->magic==1234567) ;  
    ...  
}
```



빈공간 리스트 구현

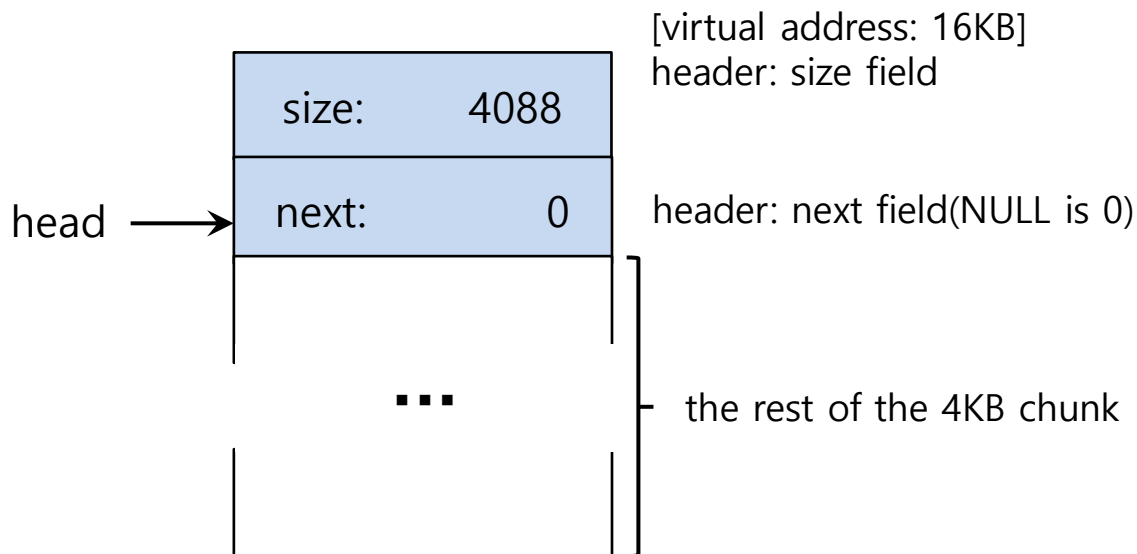
```
typedef struct __node_t {  
    int size;  
    struct __node_t *next;  
} nodet_t;
```



힙 초기화

// mmap() 함수로 사용할 메모리의 주소를 얻는다.

```
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,  
                    MAP_ANON|MAP_PRIVATE, -1, 0);  
  
head->size = 4096 - sizeof(node_t);  
head->next = NULL;
```





빈 공간 리스트 구현

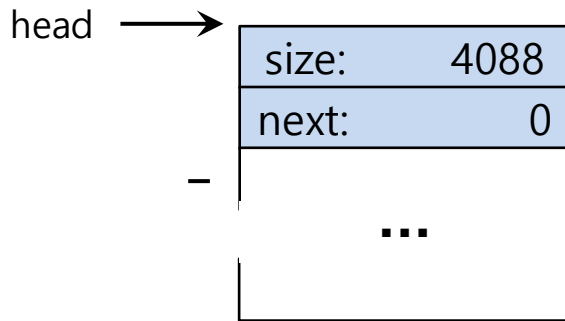
- 메모리 조각이 필요할 때, 빈 공간리스트에서 첫번째 충분히 큰 조각을 찾은 후
- 큰 조각을 둘로 나누고(split)
 - 하나는 요청용, 하나는 나머지 빈공간 관리용
 - 빈 공간용 조각의 크기를 계산해 저장한다.



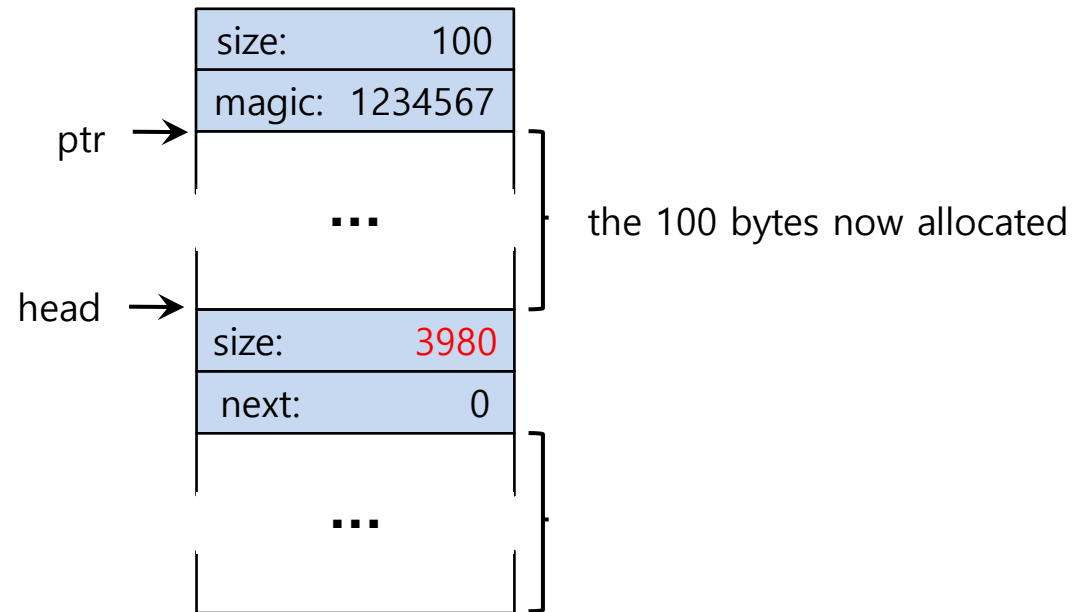
빈 공간 리스트 구현

- Example: a request for 100 bytes by `ptr = malloc(100)`

A 4KB Heap With One Free Chunk

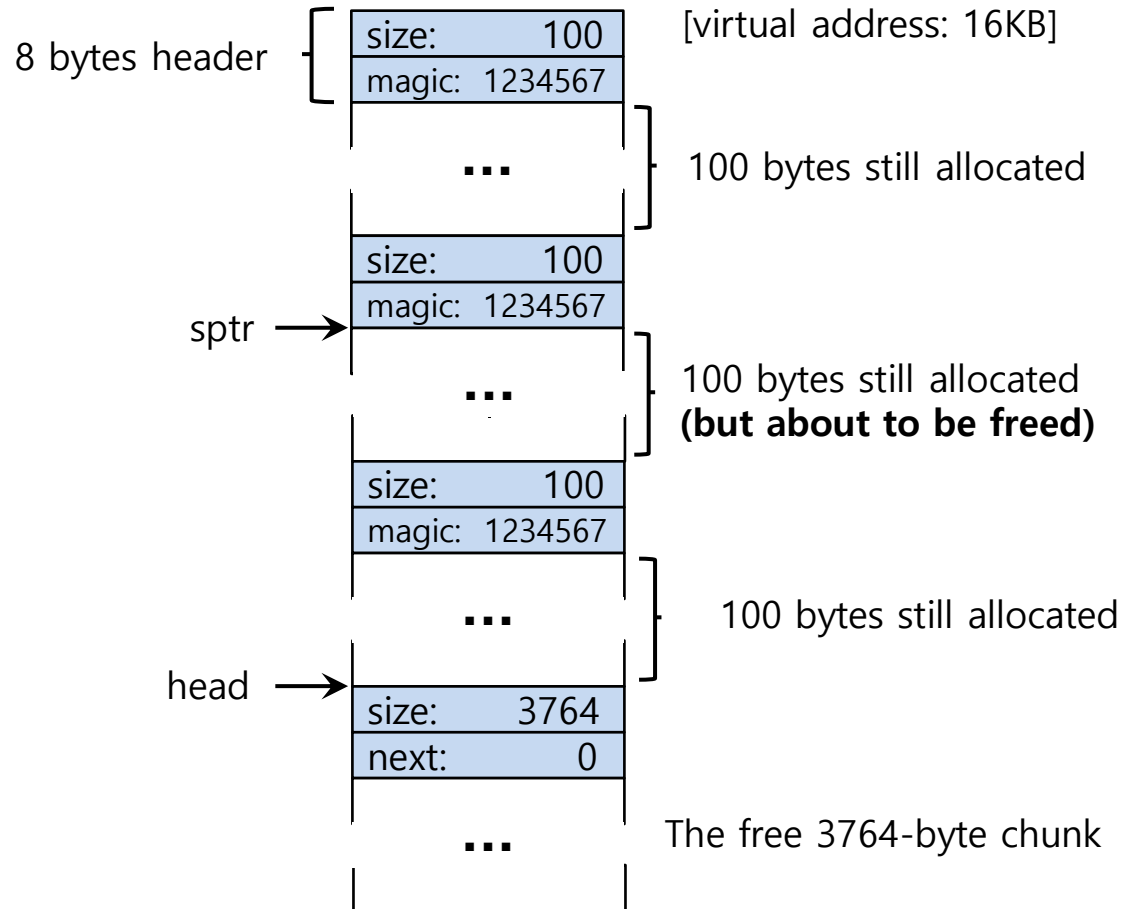


A Heap : After One Allocation





메모리 조각 할당

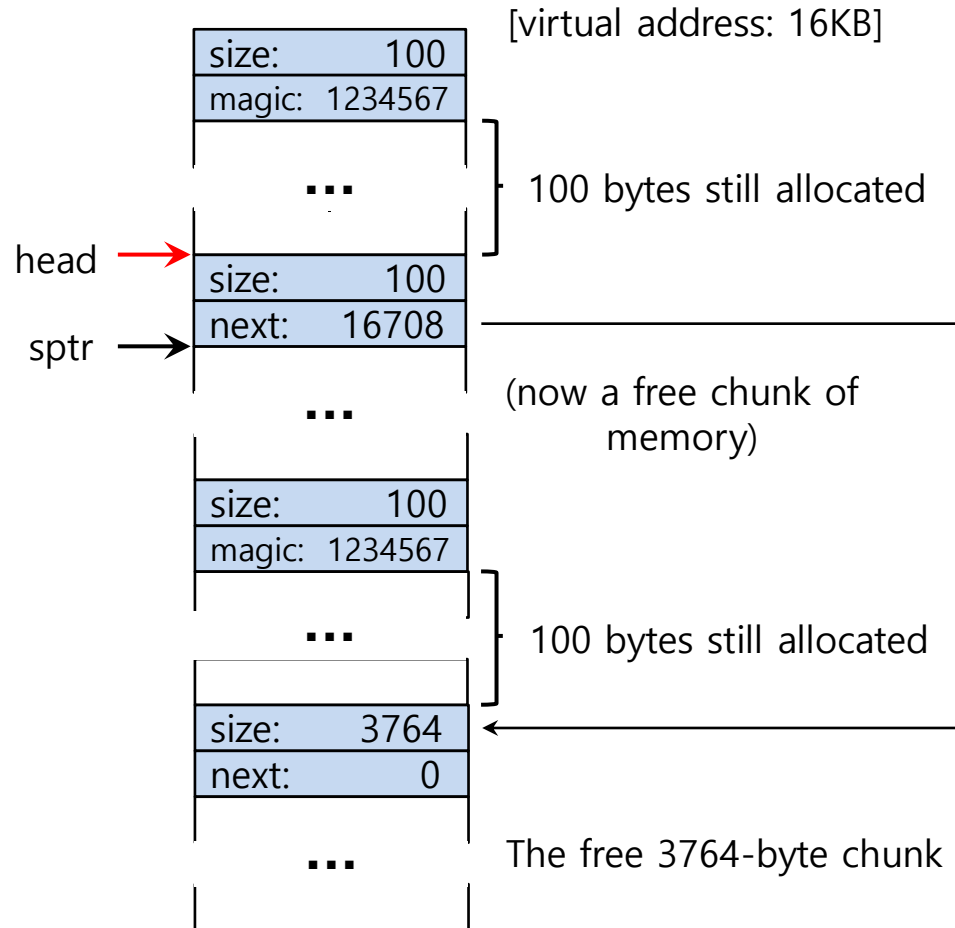


Free Space With Three Chunks Allocated



free () 를 사용한 해제

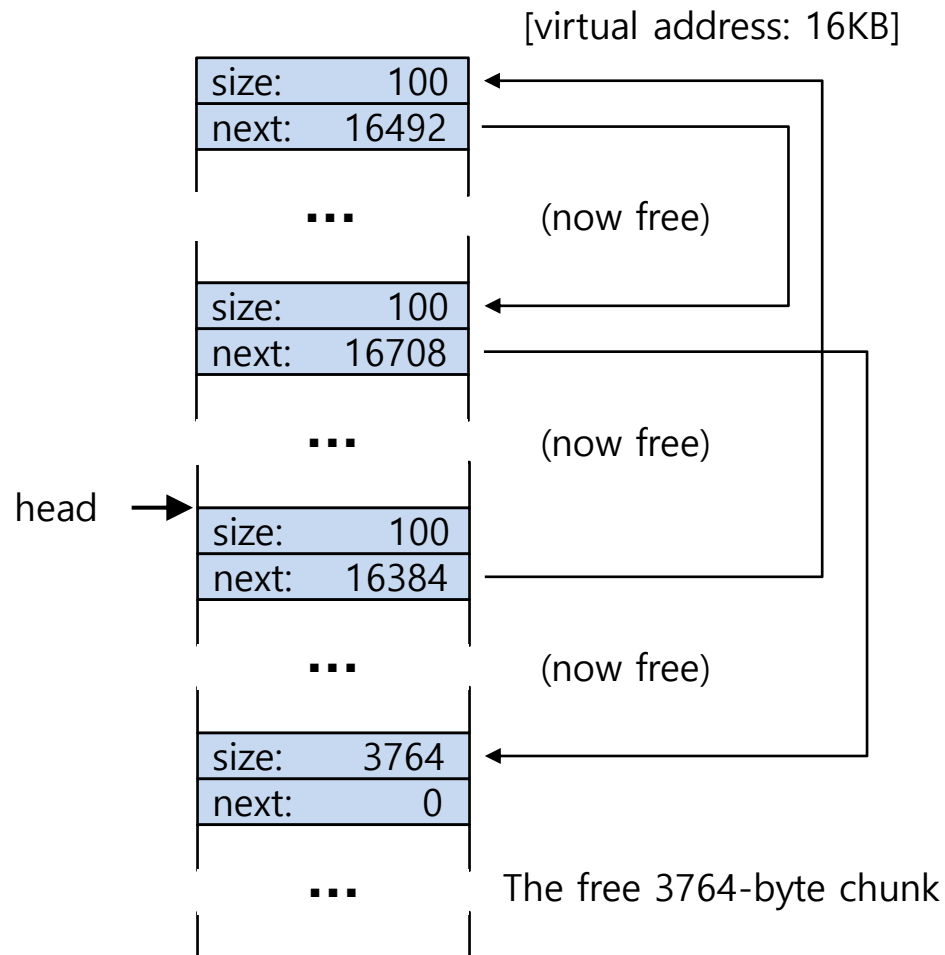
▣ free (sptr)





해제된 조각의 관리

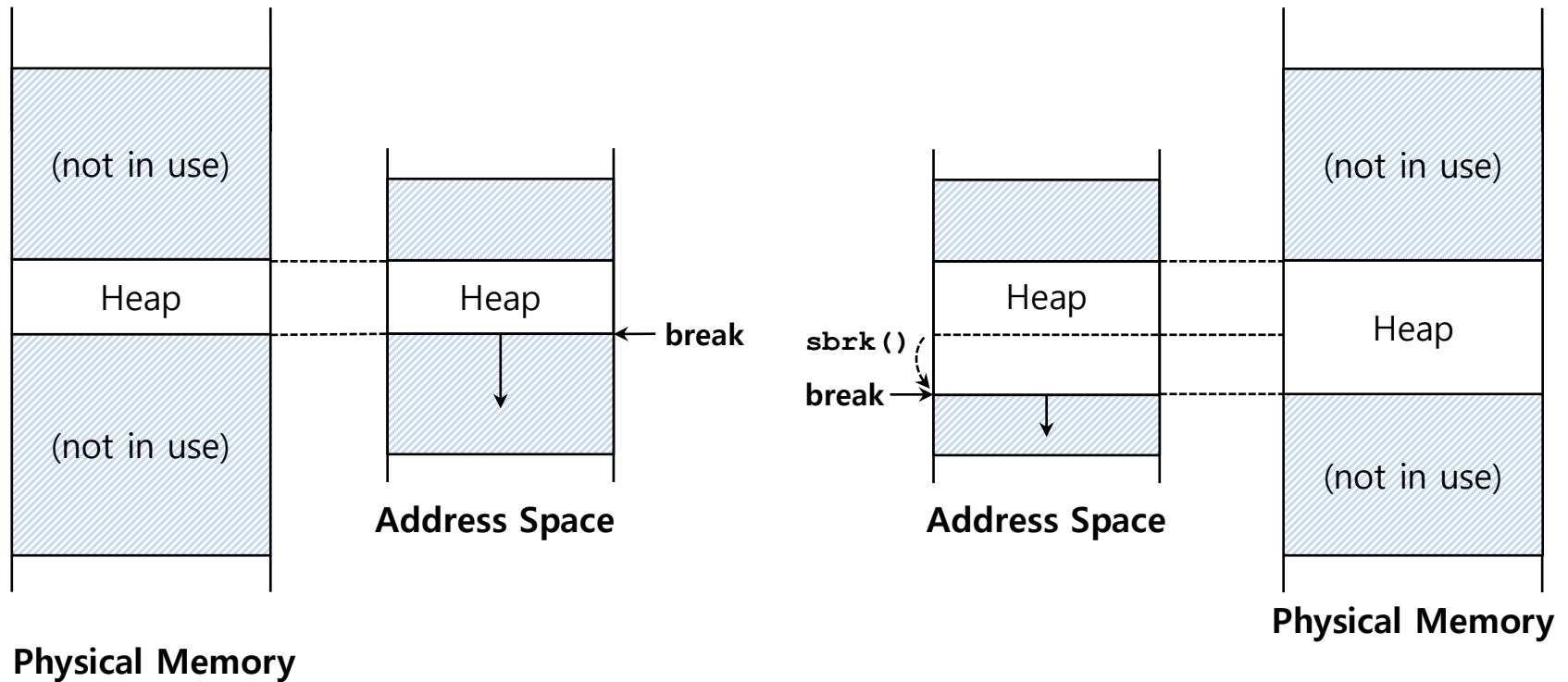
- 전부 해제되었을 경우
- 외부 단편화(External Fragmentation) 발생.
 - 병합(Coalescing)이 필요하다.





힙의 확장

- 대부분의 메모리 관리자는 처음의 작은 크기의 힙을 할당해 시작하고, 실행시 부족하면 OS에게 더 요청한다.
 - e.g., `sbrk()`, `brk()` in most UNIX systems.





빈 공간 관리: 기본 전략

- 최적 접합(Best Fit):
 - 요청보다 크거나 같은 조각들을 찾는다.
 - 조각들 중 가장 작은 것을 반환한다.
- 최악 접합(Worst Fit):
 - 요청보다 크거나 같은 조각들을 찾는다.
 - 조각들 중 가장 큰 것을 반환한다.



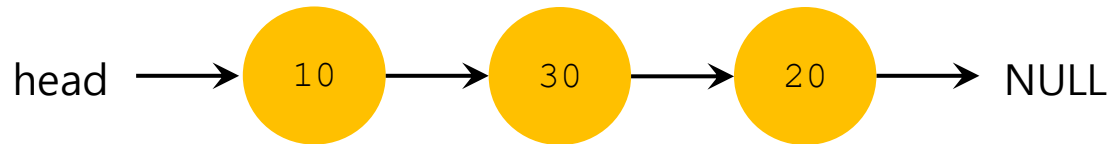
빈 공간 관리 : 기본 전략

- 최초 적합 (First Fit):
 - 처음 만나는 요청보다 크거나 작은 조각을 반환한다.
- 다음 적합 (Next Fit):
 - 지난번에 찾았던 위치 다음에 처음 만나는 요청보다 크거나 작은 조각을 반환한다.

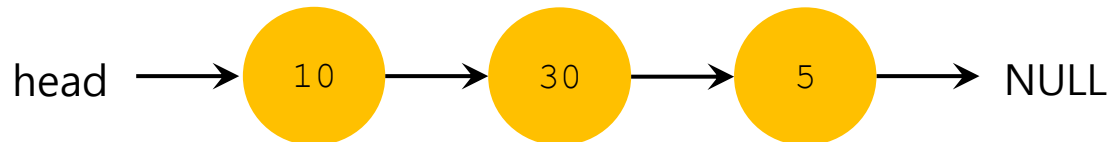


기본전략의 예

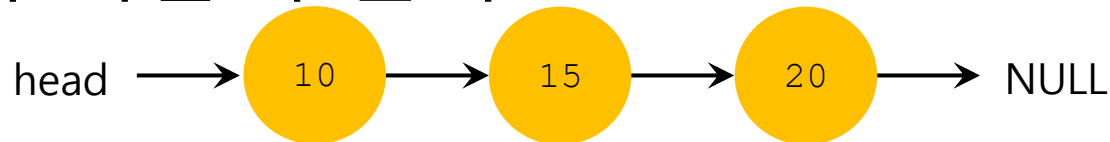
- 15바이트 요청이 있을 경우



- 최적 적합의 결과



- 최악 적합의 결과





다른 접근법: 개별 리스트

- 개별리스트:
 - 많이 사용되는 크기별로 별도의 리스트를 만들어 관리.
 - 해당 크기의 조각의 재 활용은 외부 단편화가 생기지 않는다.
 - 해당 크기의 조각의 할당은 검색이 필요 없다.
 - 새로운 문제:
 - 각 크기별로 얼마나 미리 할당해 놓아야 하는가?
 - **슬랩 할당기(Slab allocator)**를 사용한다.



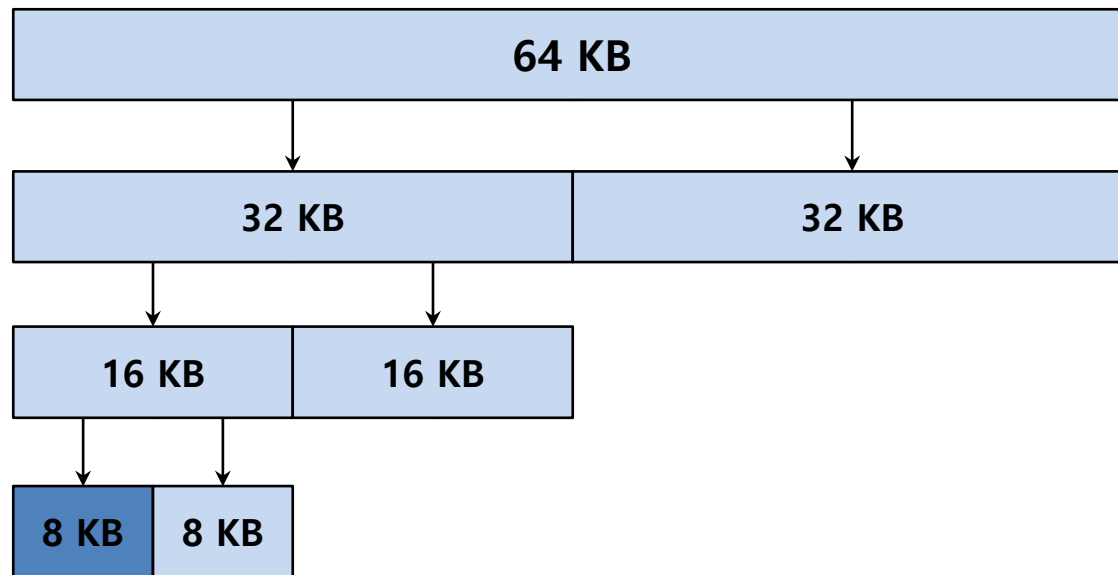
다른 접근법: 개별 리스트

- 슬랩 할당기
 - 객체별로 리스트 관리.
 - i-node, lock, PCB
 - 빈 객체들을 사전에 초기화된 상태로 유지.
 - 반납 시 초기화된 상태 유지
 - **캐시관리** : 리스트가 비어 가면 미리 한번에 여러 개를 할당 받아 저장한다.



다른 접근법: 버디 할당

- 이진 버디 할당기(Binary Buddy Allocation)
 - 맞는 사이즈의 조각이 나올 때 까지 반으로 잘라서 빈 공간 관리. (= 더 이상 자를 필요가 없을 때 까지 자른다.)



64KB 빈 공간에 7KB 요청



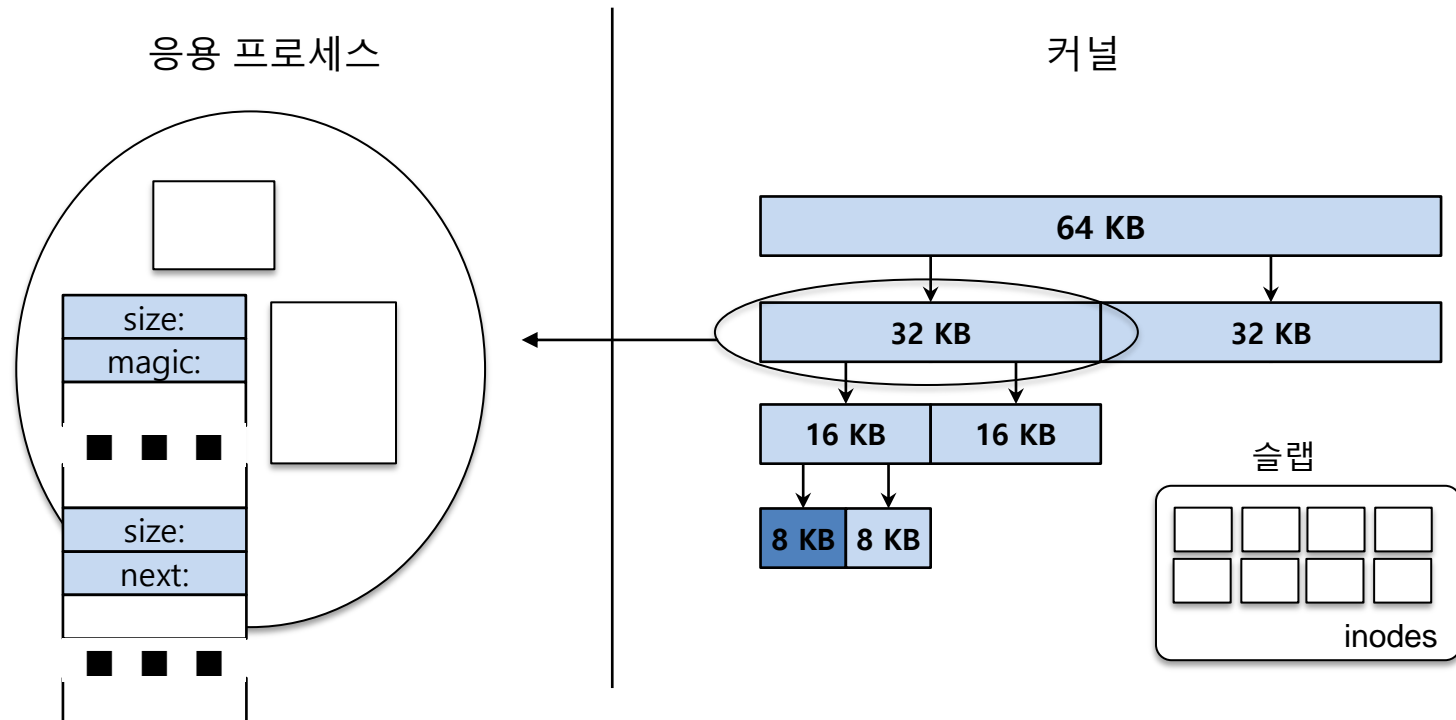
다른 접근법 : 버디 할당

- 내부 단편화가 있음.
- 외부 단편화도 있음.
- 병합 조건 존재.
 - 이웃 블록(**Buddy Block**)만 병합 가능
 - buddy : 원래 하나였던 두개.



실제 메모리 할당

- libc
 - 링크드 리스트 기반
- 커널
 - Buddy: 프로세스에게 메모리 할당
 - Slab: 커널내부에서 작은 객체 할당 (PCB, inode, socket등)



숙제 6

■ 버디 메모리 관리를 구현하여 결과를 제출하라

■ 입력 :

- 첨부한 프로젝트에 있는 memdata.txt 사용
- 메모리 요청 리스트 : 한줄당 “도착시간 크기 사용시간”으로 구성
- 요청의 끝은 <-1 -1 -1>로 표시

■ 구현 할 것

- 컴퓨터의 전체 메모리 크기는 512이다.
- BestFit, WorstFit, NextFit, Buddy로 메모리 관리를 구현해서 실행하라.
- 조건
 - 압축은 없지만 병합은 해야 한다.
 - 메모리 요청은 독립적이다. 메모리 할당에 실패해도, 메모리 요청은 계속 발생한다.
 - 메모리 요청은 FIFO로 할당된다. 실패한 메모리 할당은 메모리가 사용가능해 질 때까지 계속 대기한다. 이 때 뒤에 발생한 요청은 계속 순서대로 대기한다.
 - 전체 종료 시간과 각 요청의 평균 대기시간을 출력하라.
 - 첨부한 프로젝트에서 FirstFit의 구현을 참고 할 것.

■ 제출 (실행화일 X, .cpp화일, memdata 실행결과)