



CHUNGNAM NATIONAL UNIVERSITY



시스템 프로그래밍

강의 12. 동적 메모리 할당

교재 9 장

<http://eslab.cnu.ac.kr>



강의 일정

주	날짜	강의실 (화)	날짜	실습실 (목)
1	9월 2일	Intro	9월 4일	리눅스 개발환경 익히기
2	9월 16일	정수	9월 11일	소수 1
3	9월 23일	소수2	9월 18일	GCC & Make,
4	9월 30일	어셈1 - 데이터 이동 제어문	9월 25일	Data lab
5	10월 7일	어셈2 - 제어문	10월 2일	펜티엄 어셈블리
6	10월 14일	어셈3 - 프로시저	10월 16일	GDB
7	10월 21일	시스템 보안	10월 23일	Binary bomb1
8	10월 28일	시험휴강	10월 30일	Binary bomb 2
9	11월 4일	프로세스 1	11월 6일	Tiny shell 1
10	11월 11일	프로세스 2	11월 13일	Tiny shell 2
11	11월 18일	시그널	11월 20일	Tiny shell 3
12	11월 25일	동적메모리 1	11월 27일	Malloc lab1
13	12월 2일	동적메모리 2	12월 4일	Malloc lab2
14	12월 12일	기말고사	12월 11일	Malloc lab3
15	12월 16일	Wrap-up/종강		



전달사항

■ 다음주 예습

- 9.9.13~9.9.14, 869-872

■ 기말고사

- 12월 11일 목요일 저녁 7시

- 02반 411호, 03반 413호



메모리에 관한 불편한 진실

프로그래머에게 메모리에 대한 이해가 중요하다

- 메모리는 무한의 자원이 아니다
 - 메모리는 할당되고 관리되는 자원이다
 - 많은 응용프로그램들은 메모리에 큰 영향을 받는다
 - ▶ 특히 복잡한 그래프 알고리즘을 사용하는 경우
- 메모리 참조 버그는 매우 치명적이다
 - 버그의 영향이 시공간 적으로 동떨어져서 발견된다
- 메모리 성능은 일정하지 않다
 - 캐시와 가상 메모리는 프로그램의 성능에 매우 영향을 줄 수 있다
 - 메모리 시스템의 특성을 반영하는 프로그램은 속도를 상당히 개선할 수 있다.

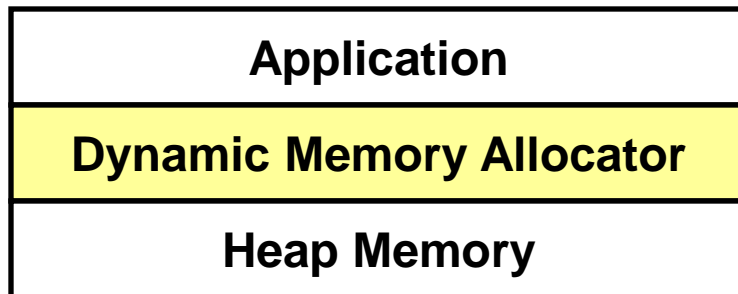


동적 메모리 할당 사용의 이유

- 프로그램이 실행되기 전에는 크기를 알 수 없는 자료 구조를 위해 사용
- 예 : n 개의 문자를 화면에서 읽어 들어서 배열에 저장하고자 할 때, n 과 문자를 차례로 받아 들어서 실행하는 경우
 - 고정 크기의 배열로도 구현 가능
 - `int array[MAX_SIZE];`
 - 이와 같이 배열의 크기를 알 수 없을 때, 최대값으로 배열을 구현하는 것은 나쁜 생각
 - ▶ 시스템의 가능한 메모리 사용량을 알 수 없다
 - ▶ MAX_SIZE 보다 많은 입력을 원한다면?
 - ▶ MAX_SIZE 값을 계속 바꿔서 다시 컴파일 해야 한다
 - 코드 관리 차원에서 안 좋음
 - 이런 경우에 동적 메모리 할당이 효과적이고, 중요한 프로그래밍 기술임



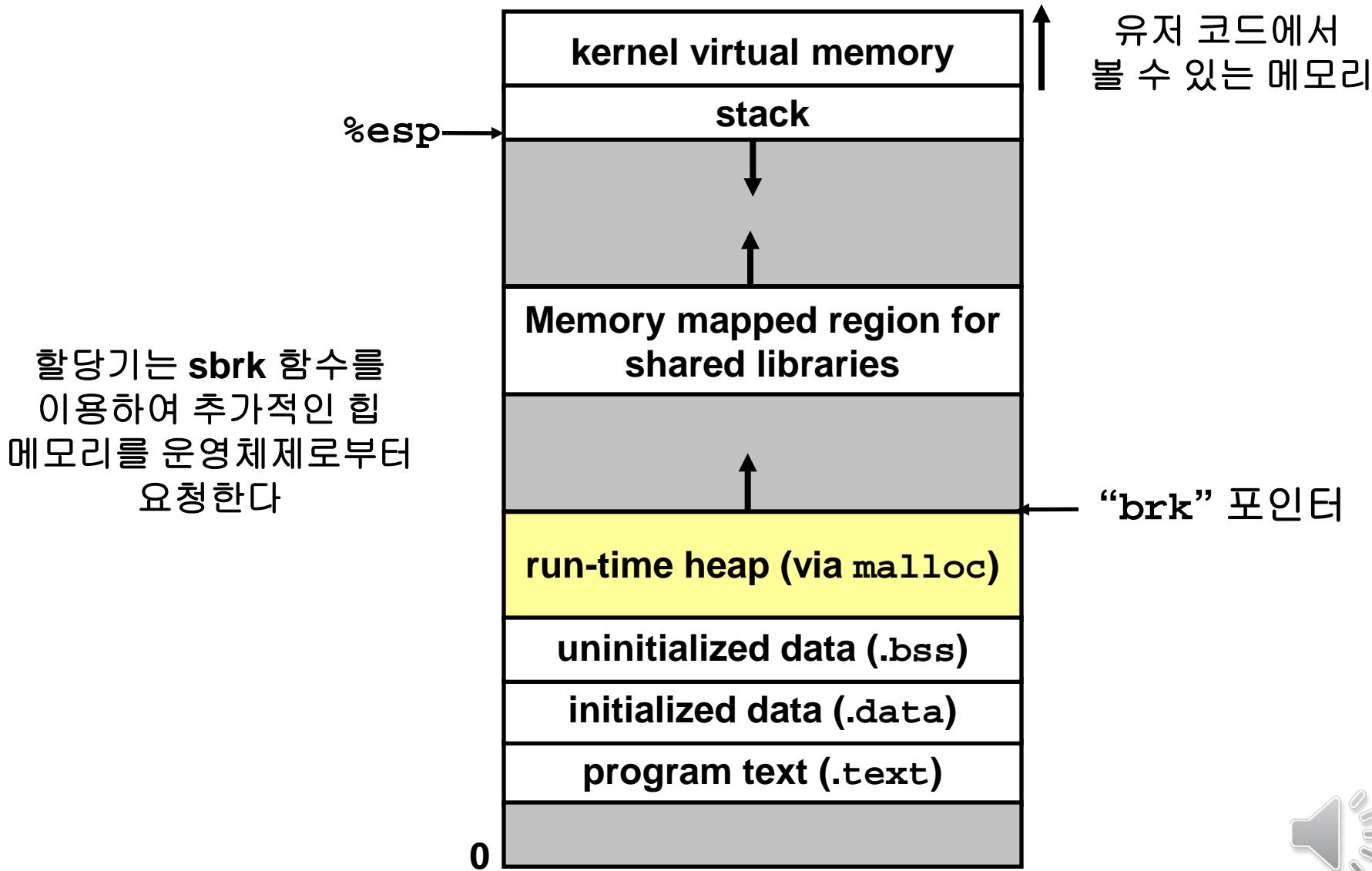
동적 메모리 할당



- 직접 vs 간접(Explicit vs. Implicit) 메모리 할당기
 - 직접 할당 : 응용프로그램이 할당하고, 반환한다
 - ▶ E.g., malloc 과 free
 - 간접 할당 : 응용프로그램이 할당하지만, 반환하지는 않는다
 - ▶ E.g. 자바에서의 가비지컬렉션
- 할당방법
 - 두가지 경우에 모두 메모리 할당기는 메모리를 블록단위로 제공한다
 - 응용프로그램에 **free** 메모리 블록을 나눠준다
- 여기서는 직접메모리 할당을 다룬다



프로세스의 메모리 이미지



Malloc 패키지

- `#include <stdlib.h>`
- `void *malloc(size_t size)`
 - 성공시 :
 - ▶ 최소 `size` 바이트의 메모리 블록의 포인터를 (대개 8바이트 단위로 맞추어) 반환
 - ▶ 만일 `size == 0`, returns NULL
 - 실패시 : **NULL (0)**을 리턴하고 `errno`를 세팅
- `void free(void *p)`
 - 가용 메모리 풀을 가리키는 블록 포인터 `p`를 리턴
 - `p`는 이전 `malloc` 이나 `realloc`에서 제공
- `void *realloc(void *p, size_t size)`
 - 블록 `p`의 크기를 변경하고, 새 블록의 포인터를 리턴
 - 새블록의 내용은 이전블록과 새 블록 크기중 적은 크기까지는 변화없음



Malloc Example

```
void foo(int n, int m) {
    int i, *p;

    /* allocate a block of n ints */
    p = (int *)malloc(n * sizeof(int));
    if (p == NULL) {
        perror("malloc");
        exit(0);
    }
    for (i=0; i<n; i++) p[i] = i;

    /* add m bytes to end of p block */
    if ((p = (int *) realloc(p, (n+m) * sizeof(int))) == NULL) {
        perror("realloc");
        exit(0);
    }
    for (i=n; i < n+m; i++) p[i] = i;

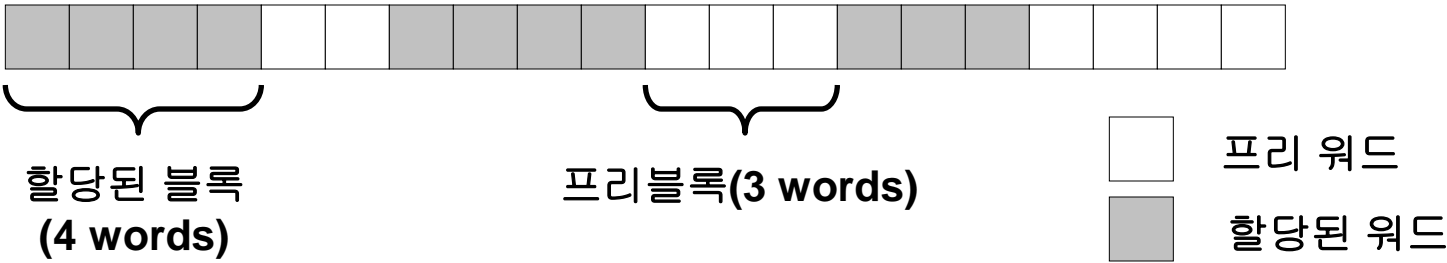
    /* print new array */
    for (i=0; i<n+m; i++)
        printf("%d\n", p[i]);

    free(p); /* return p to available memory pool */
}
```



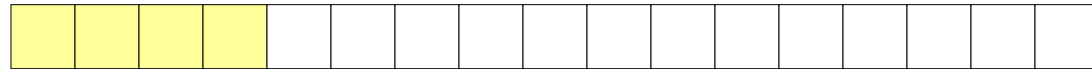
가정

- 메모리는 워드 단위로 주소가 지정된다

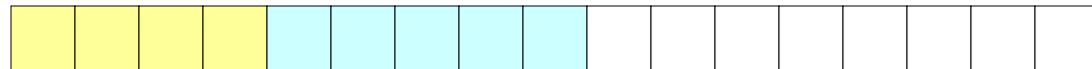


할당 예제

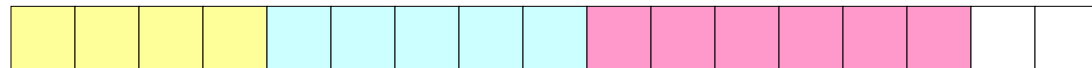
```
p1 = malloc(4)
```



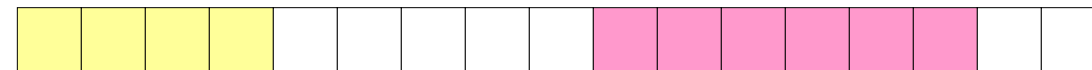
```
p2 = malloc(5)
```



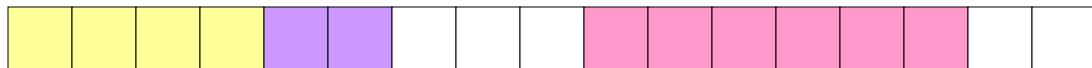
```
p3 = malloc(6)
```



```
free(p2)
```



```
p4 = malloc(2)
```



제한사항

■ 응용 프로그램

- 무순의 할당과 반환 요청이 발생된다
- 반환 요청은 할당된 블록에 대응되어야 한다

■ 할당 프로그램

- 할당된 블록의 갯수와 크기는 조정할 수 없다
- 모든 할당 요청에 즉시 반응해야 한다
 - ▶ 즉, 요청 순서를 변경하거나 요청을 버퍼에 저장할 수 없다
- 블록은 프리메모리에서 할당해야 한다
 - ▶ 즉, 할당된 블록을 프리 메모리에만 배치할 수 있다
- 블록들은 모든 주소맞춤 요건을 만족하도록 정렬해야 한다
 - ▶ 리눅스 컴퓨터의 GNU malloc에서는 8바이트 정렬사용
- 프리 메모리만을 관리하고 수정할 수 있다
- 일단 할당된 블록들은 이동할 수 없다
 - ▶ 즉, 압축은 사용할 수 없다



우수한 malloc/free 프로그램의 목표

■ 주요 목표

- malloc 과 free 에서 우수한 시간성능을 얻는다
 - ▶ 이상적으로는 상수시간이 걸려야 한다(언제나 가능한 것은 아니다)
 - ▶ 당연히 블록의 수에 비례하는 시간이 걸리면 안된다
- 우수한 공간 이용율을 가져야 한다
 - ▶ 유저가 할당한 구조체는 힙의 많은 부분을 차지해야 한다
 - ▶ 단편화“fragmentation”를 최소화해야 한다

■ 다른 목표

- 우수한 지역성
 - ▶ 시간적으로 인접해서 할당된 구조체는 공간상으로 인접해야 한다
 - ▶ 유사한 객체들은 인접공간에 할당되어야 한다
- 견고성
 - ▶ free(p1) 함수가 유효한 포인터 p1에 대해 수행하는지 체크할 수 있다
 - ▶ 메모리 참조가 할당된 공간으로 이루어지는지 체크할 수 있다



성능 지표 1 : 처리량 Throughput

- malloc 과 free 요청이 다음과 같이 주어진다고 하자
 - $R_0, R_1, \dots, R_k, \dots, R_{n-1}$
- 처리량과 순간 최대 메모리 사용율을 극대화하고자 한다
 - 이 목표들은 종종 충돌한다
- 처리량 Throughput:
 - 단위 시간동안에 완료한 요청의 수
 - 예.
 - ▶ 5,000번의 malloc 과 5,000번의 free 를 10 초 동안에 수행하는 경우
 - ▶ 작업량 = 1,000 operations/second.
 - alloc과 free 함수의 평균 처리 시간을 최소화하면 throughput을 극대화 할 수 있다



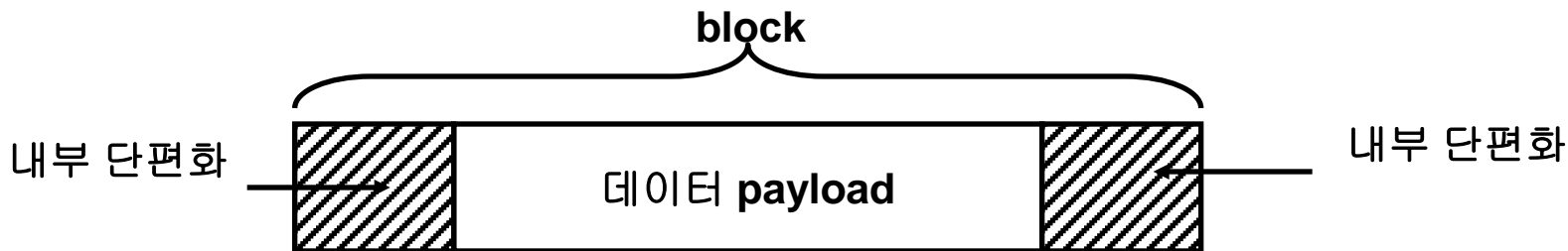
성능지표 2 : 최대 메모리 이용율

- 좋은 프로그래머는 가상메모리의 크기도 제한되어 있다는 것을 알고 작업한다
 - 따라서 효율적으로 관리해야 한다
- 효율적인 heap 사용 지표 : peak heap utilization
- malloc과 free 요청의 순서가 다음과 같이 주어질때
 - $R_0, R_1, \dots, R_k, \dots, R_{n-1}$
- Def: 총 데이터 P_k :
 - `malloc(p)` 는 p 바이트의 데이터를 포함하는 메모리 블록을 리턴
 - 요청 R_k 가 처리된 후에, 전체 데이터 P_k 는 현재 할당된 데이터들의 합이다
- Def: 현재 힙의 크기는 H_k
 - H_k 는 단조 증가한다고 가정
- Def: 최대 메모리 이용율:
 - k개의 요청 후에, 최대 메모리 이용율은:
 - ▶ $U_k = (\max_{i \leq k} P_i) / H_k$
- 구현 목표 : U_{n-1} 을 가능한 모든 작업 순서에 대해서 극대화한다



내부 메모리 단편화(Internal Fragmentation)

- 나쁜 메모리 사용률은 단편화로 인해 발생
 - 두 가지 형태로 발생 : 내부 및 외부 단편화
- 내부 단편화 Internal fragmentation
 - 일부 블록에서 내부 단편화는 블록 크기와 데이터 크기간의 차이로 인해 발생



- 힙 자료구조를 유지하고, 정렬을 위해 바이트를 추가하거나 할당 정책에 의한 오버헤드 (블록을 나누지 않는 경우) 때문에 발생
- 이전 요청 패턴에 의해서만 영향을 받으므로 측정이 용이.



외부 메모리 단편화(External Fragmentation)

힙 전체 메모리를 합쳐보면 수용이 가능하지만 가용(**free**) 블록 하나의 크기가 작은 경우에 발생

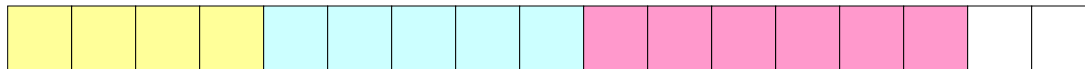
```
p1 = malloc(4)
```



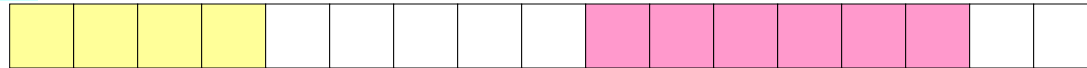
```
p2 = malloc(5)
```



```
p3 = malloc(6)
```



```
free(p2)
```



```
p4 = malloc(6)
```

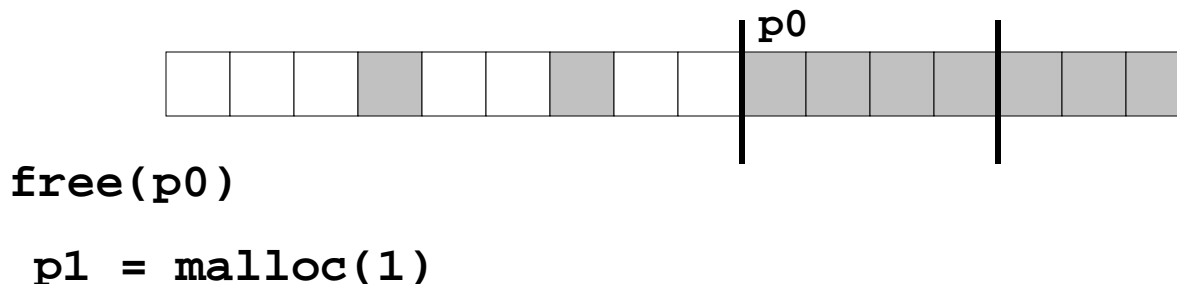
oops!

외부 단편화는 미래의 요청패턴에 의해 결정되므로, 측정이 어렵다



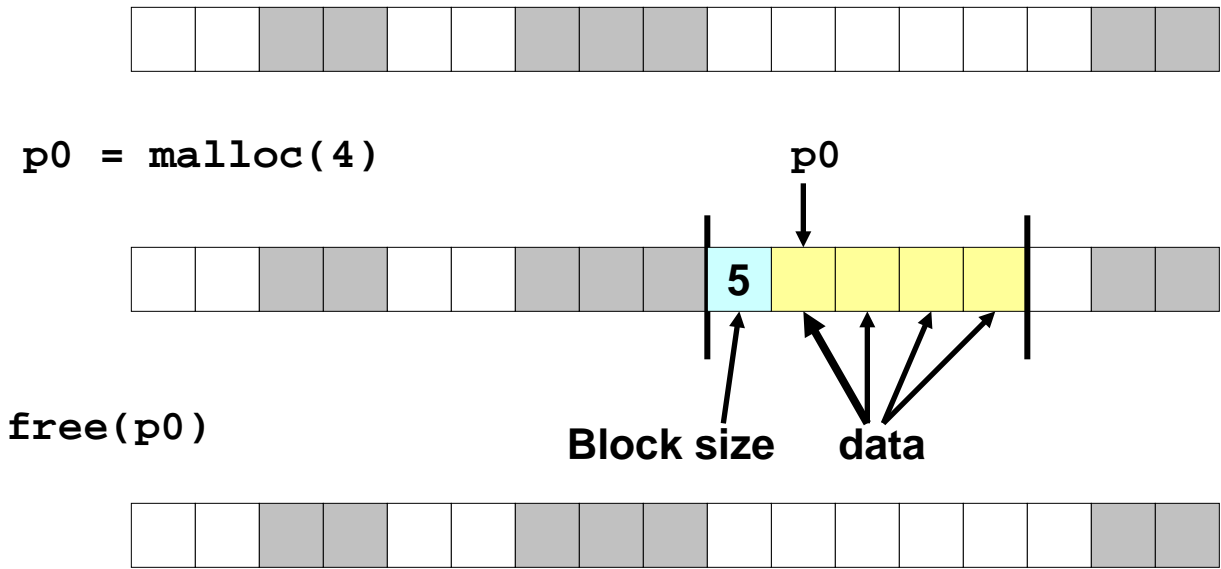
구현시 고려할 점

- 포인터 한개로 부터 얼마 만큼의 메모리를 반환해야하는 지를 어떻게 알 수 있을까?
- 가용블록을 어떻게 관리할 것인가?
- 가용블록보다 작은 크기의 구조체를 할당할 때, 남는 공간은 어떻게 할 것인가?
- 할당을 위한 블록은 어떻게 선택하는가 – 여러 블록에 할당이 가능한 경우에?
- 반환된 블록을 다시 가용블록으로 어떻게 관리하는가?



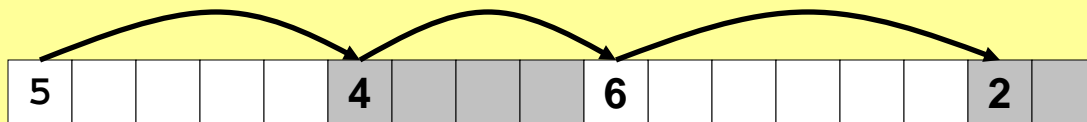
얼마만 큼 Free 시켜야 하는지 결정하기

- 일반적인 방법
 - 블록 앞에 블록의 크기를 저장한다.
 - ▶ 이 워드를 헤더라고 한다
 - 매 할당 블록마다 추가적으로 1워드가 소요된다

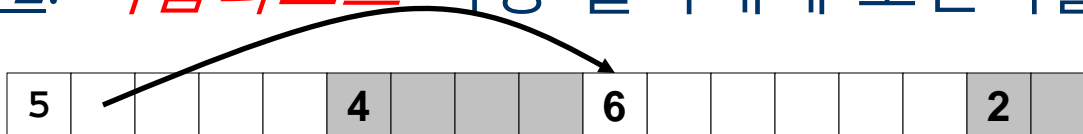


Free 블록 관리하기

- 방법 1: 간접리스트 크기 정보를 이용하여 모든 블록을 연결



- 방법 2: 직접리스트 가용 블록내에 포인터를 이용



- 방법 3: 구분 가용 리스트 segregated free list

- 크기 클래스마다 각각 별도의 가용 리스트를 유지

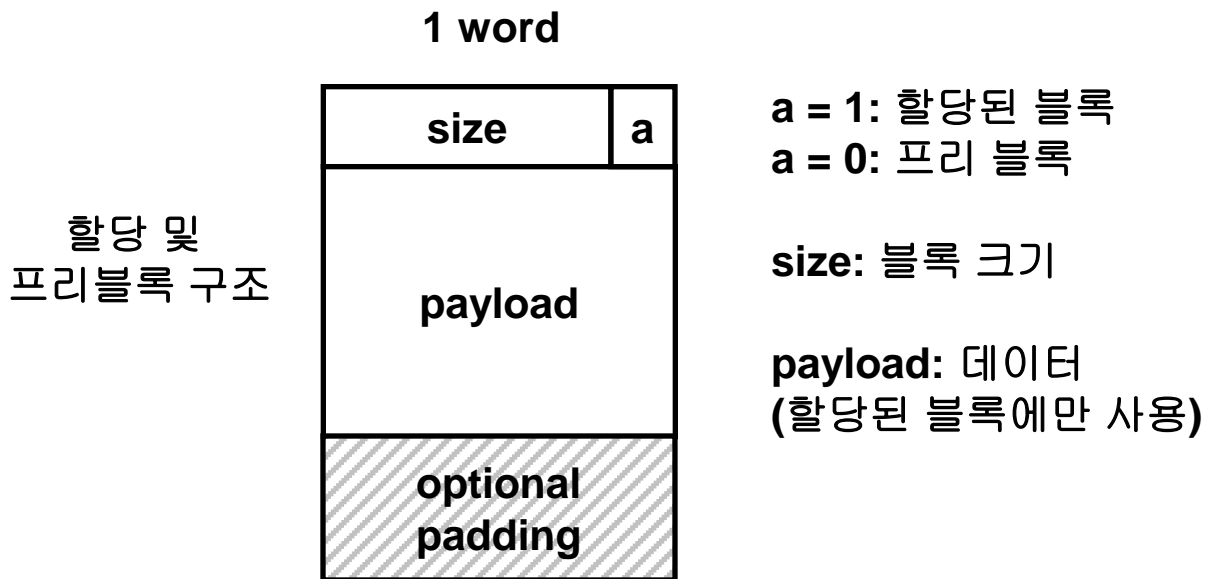
- 방법 4: 크기로 정렬된 블록

- 가용 블록내에 포인터를 이용하고, 크기를 키로 사용하여 균형 트리를 사용할 수 있다



방법 1: 간접 리스트 방식(Implicit List)

- 각 블록이 비어있는지 할당상태인지 구분할 필요가 있다
 - 추가적인 비트를 사용
 - 블록 크기가 항상 2의 배수라면 헤더에 함께 이용할 수 있다 (크기 헤더를 읽을 때 비트를 제거).



연습문제 1

- 다음과 같은 순서로 malloc 요청을 하는 경우에 블록의 크기와 header에 저장되는 값을 쓰시오. 헤더는 4바이트를 사용한다.
- 단, 할당은 double-word alignment를 유지해야 하고, header만 사용하는 간접리스트 방식으로 메모리를 관리한다고 가정한다.
 - 할당되는 블록의 크기는 8의 배수로 해야 한다.

요청	블록크기(바이트)	블록 header(hex)
Malloc(1)		
Malloc(5)		
Malloc(12)		
Malloc(13)		



간접 리스트 : Free 블록 찾기

■ 최초할당 *First fit*:

- 처음부터 검색 시작해서 맨처음 크기가 맞는 블록을 할당

```
p = start;
while ((p < end) &&          \\ not passed end
      ((*p & 1) ||          \\ already allocated
      (*p <= len)))         \\ too small
    p = p + (*p & -2);       \\ goto next block
```

- 모든 블록(할당 및 가용블록 포함)의 수에 비례해서 시간이 소요
- 실제로는 리스트 시작부분에 작은 조각들이 다수 발생할 수 있다

■ 다음할당 *Next fit*: (Donald Knuth 에 의해 제안됨)

- first-fit과 유사하지만, 이전 검색이 종료된 위치에서 검색을 시작
- 연구결과 단편화 성능은 더 나쁨

■ 최적할당 *Best fit*:

- 리스트를 검색해서 가장 근접한 크기의 블록을 선택
- 조각을 작게 해준다 – 대부분 단편화를 개선해준다
- 대개 최초할당보다 느리게 동작한다

■ 세가지 검색 방법을 비교할 수 있는가?



비트 필드 용법

■ Header 표시방법 :

■ 마스크와 비트 연산자

```
#define SIZEMASK                (~0x7)
#define PACK(size, alloc)      ((size) | (alloc))
#define getSize(x)              ((x)->size & SIZEMASK)
```

■ 비트 필드 bitfields

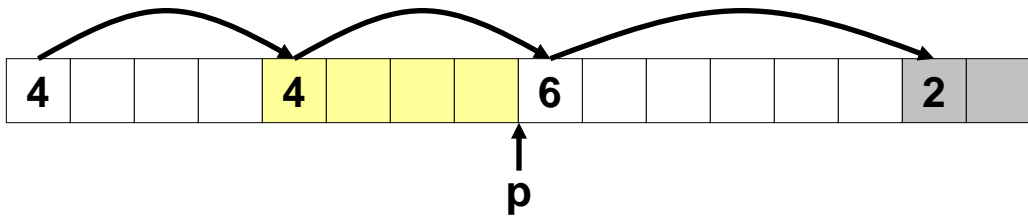
```
struct
{
    unsigned allocated:1;
    unsigned size:31;
} Header;
```



간접 리스트 : Free 블록 할당

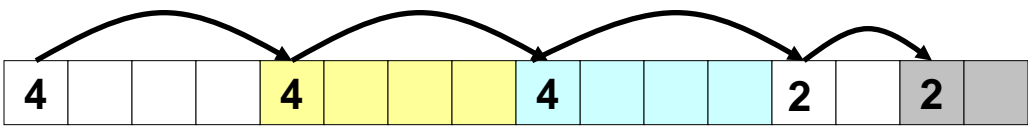
가용블록에서 할당하기 - 쪼개기

- 할당된 공간이 가용한 공간보다 크기가 더 작을 수 있으므로, 이 블록을 쪼갬다



```
void addblock(ptr p, int len) {
    int newsize = ((len + 1) >> 1) << 1; // add 1 and round up
    int oldsize = *p & -2;                // mask out low bit
    *p = newsize | 1;                     // set new length
    if (newsize < oldsize)
        *(p+newsize) = oldsize - newsize; // set length in remaining
                                           // part of block
}
```

addblock(p, 4)



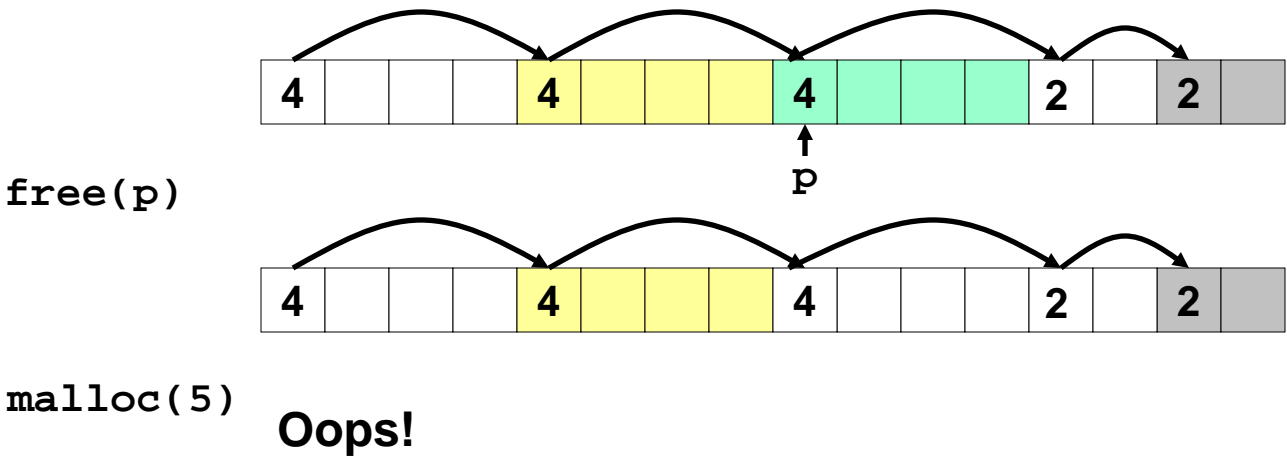
간접 리스트 : 블록 free 시키기

가장 간단한 구현:

- 할당 플래그만 0으로 세팅

```
void free_block(ptr p) { *p = *p & -2 }
```

- 하지만, 잘못된 단편화가 발생할 수 있다



가용공간이 있음에도 불구하고, 할당기는 할당할 블록을 찾지 못한다

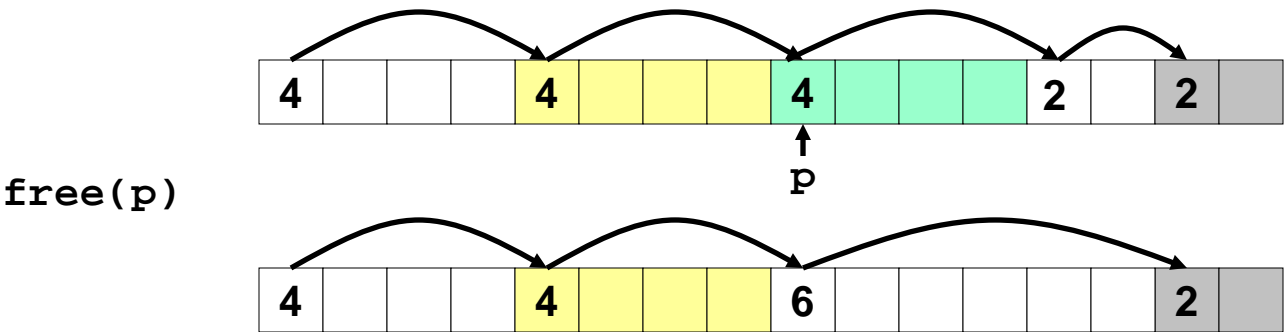


간접 리스트 : 결합(Coalescing)

■ 다음 또는 이전 블록이 free 하면 함께 연결해서 더 큰 free 블록을 만든다

● 다음 블록과 결합하기

```
void free_block(ptr p) {
    *p = *p & -2;           // clear allocated flag
    next = p + *p;           // find next block
    if ((*next & 1) == 0)
        *p = *p + *next;     // add to this block if
                             // not allocated
}
```



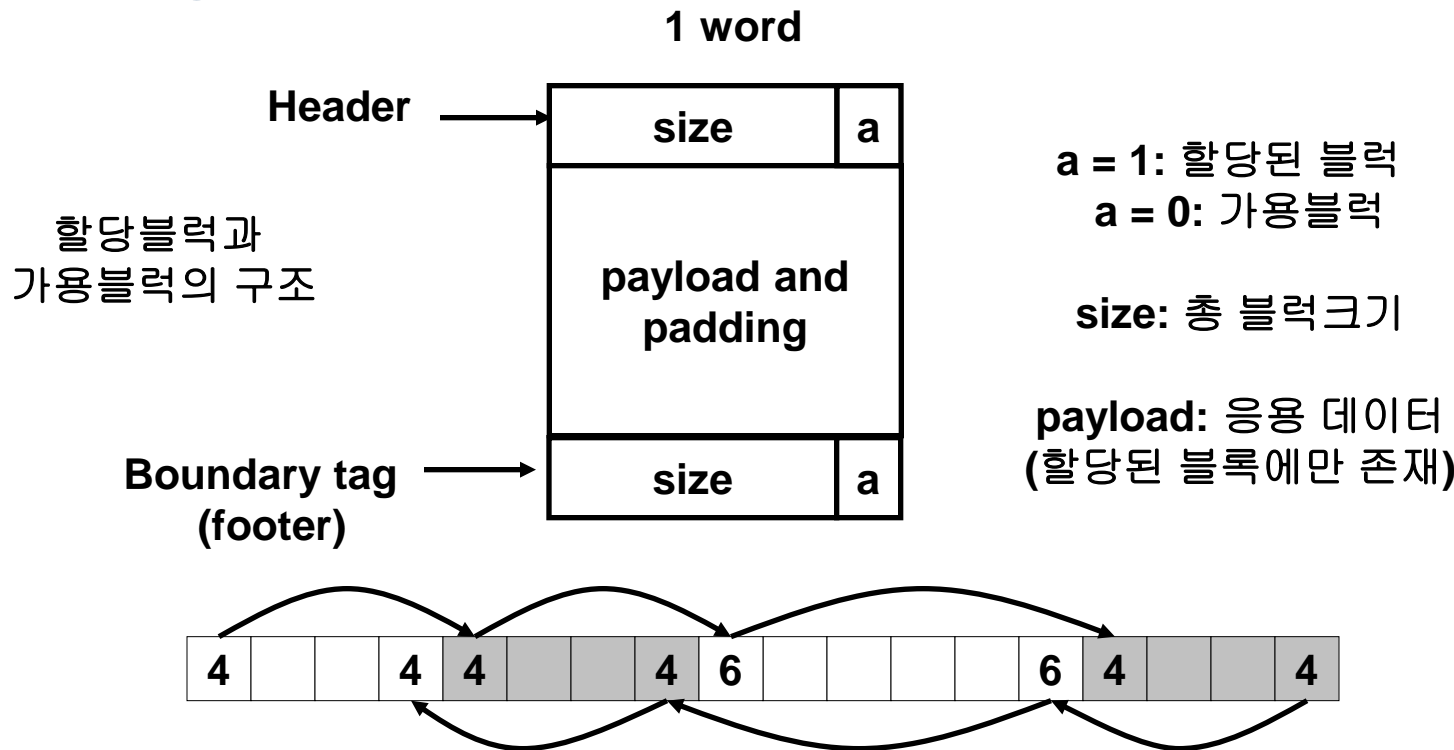
● 하지만 앞 블록과는 어떻게 결합할 수 있는가?



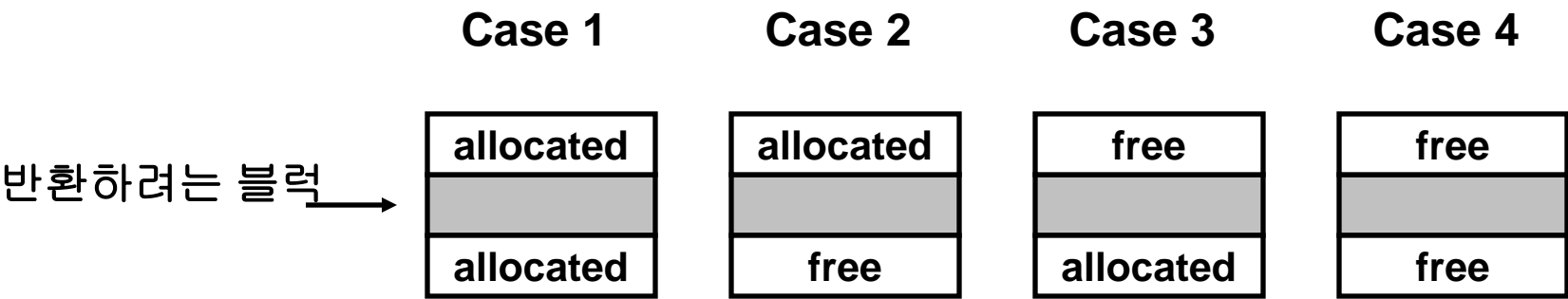
간접 리스트 : 양방향 연결

■ *경계 태그 Boundary tags* [Knuth73]

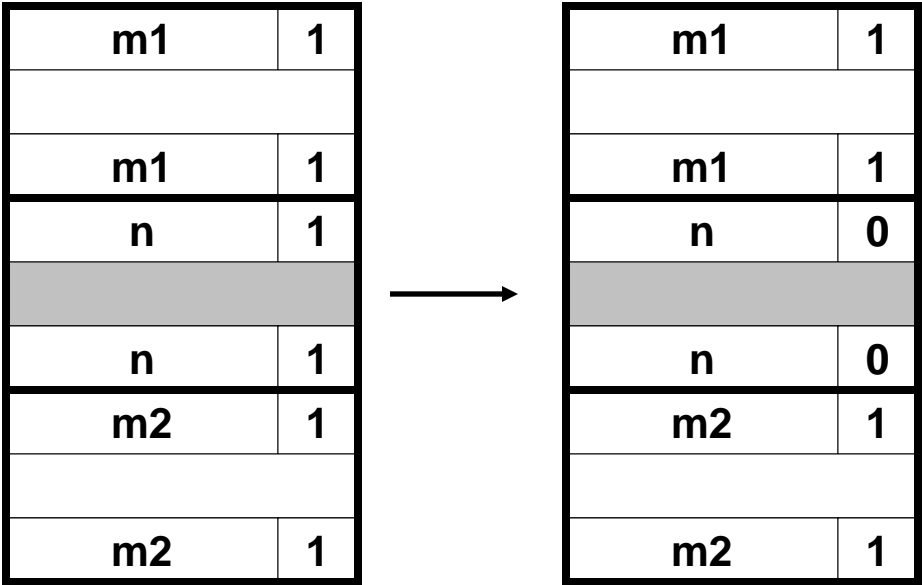
- 프리한 블록의 마지막에 헤더를 복사해서 사용
- 리스트를 거꾸로 따라갈 수 있게 해준다. 그러나, 추가적인 공간이 소요된다
- 중요하고 일반적인 기법!



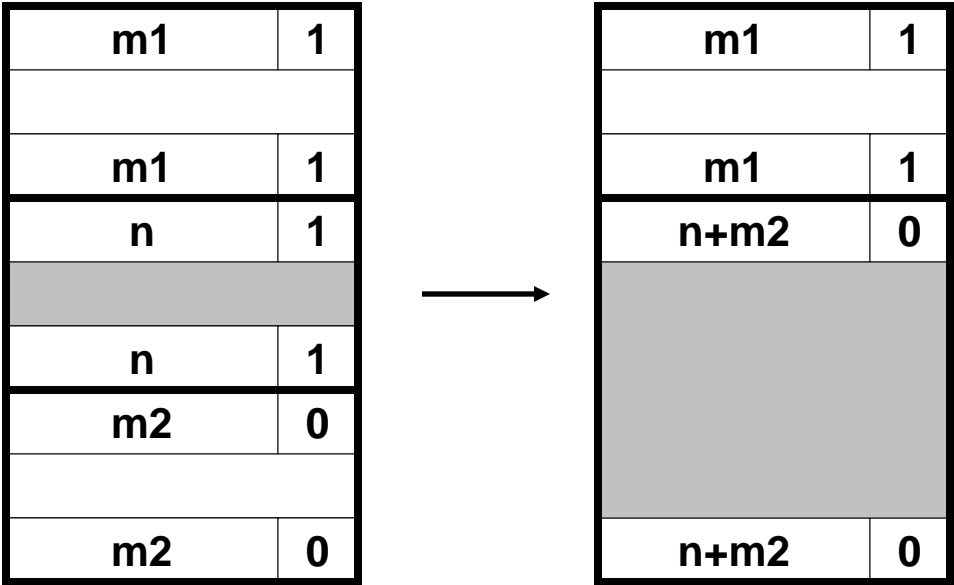
상수 소요시간 연결법



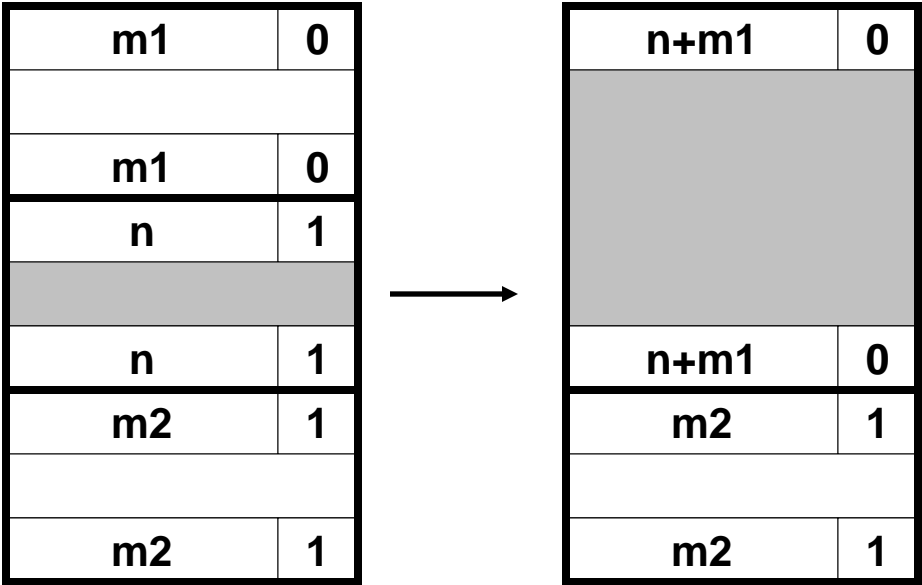
상수시간 연결법 (Case 1)



상수시간 연결법 (Case 2)



상수시간 연결법 (Case 3)



상수시간 연결법 (Case 4)

