CHUNGNAM NATIONAL UNIVERSITY

시스템 프로그래밍

강의 12. 메모리
교재 9 장
Nov. 19, 2010
http://eslab.cnu.ac.kr

전달 사항

- 다음주 화요일 수업 휴강 => 오늘 보강했으니

학습 내용

- 제 1주 : 과목 소개 및 시스템 개요
- 제 2주 ~ 3주 : 컴퓨터에서의 데이터 표현
  - 정수
  - 실수
- 제 4주~7주 : 어셈블리어와 프로세서 구조
  - 데이터의 이동
  - 제어문
  - 프로시져 및 스택프레임
  - 버퍼 오버플로우
- 제9주 ~ 10주 예외적 인 제어흐름
  - 프로세스
  - 시그널
- 제 11주 입출력 시스템
- 제 12 ~ 13주 메모리 관리
  - 가상메모리
  - 동적 메모리 할당
- 제 14주 동시성 프로그래밍 (Concurrent programming)

메모리에 관한 불편한 진실

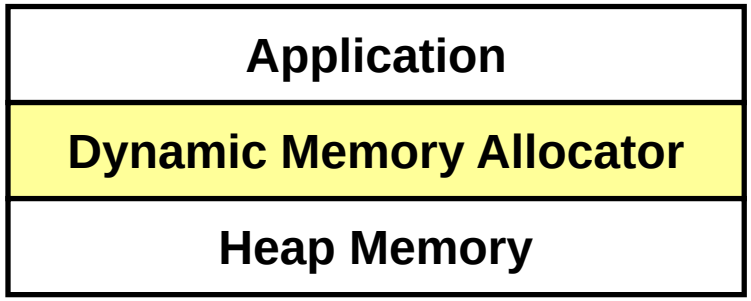
Memory Matters

- Memory is not unbounded
  - **It must be allocated and managed**
  - **Many applications are memory dominated**
    - Especially those based on complex, graph algorithms
- Memory referencing bugs especially pernicious
  - **Effects are distant in both time and space**
- Memory performance is not uniform
  - **Cache and virtual memory effects can greatly affect program performance**
  - **Adapting program to characteristics of memory system can lead to major speed improvements**

동적 메모리 할당 사용의 이유

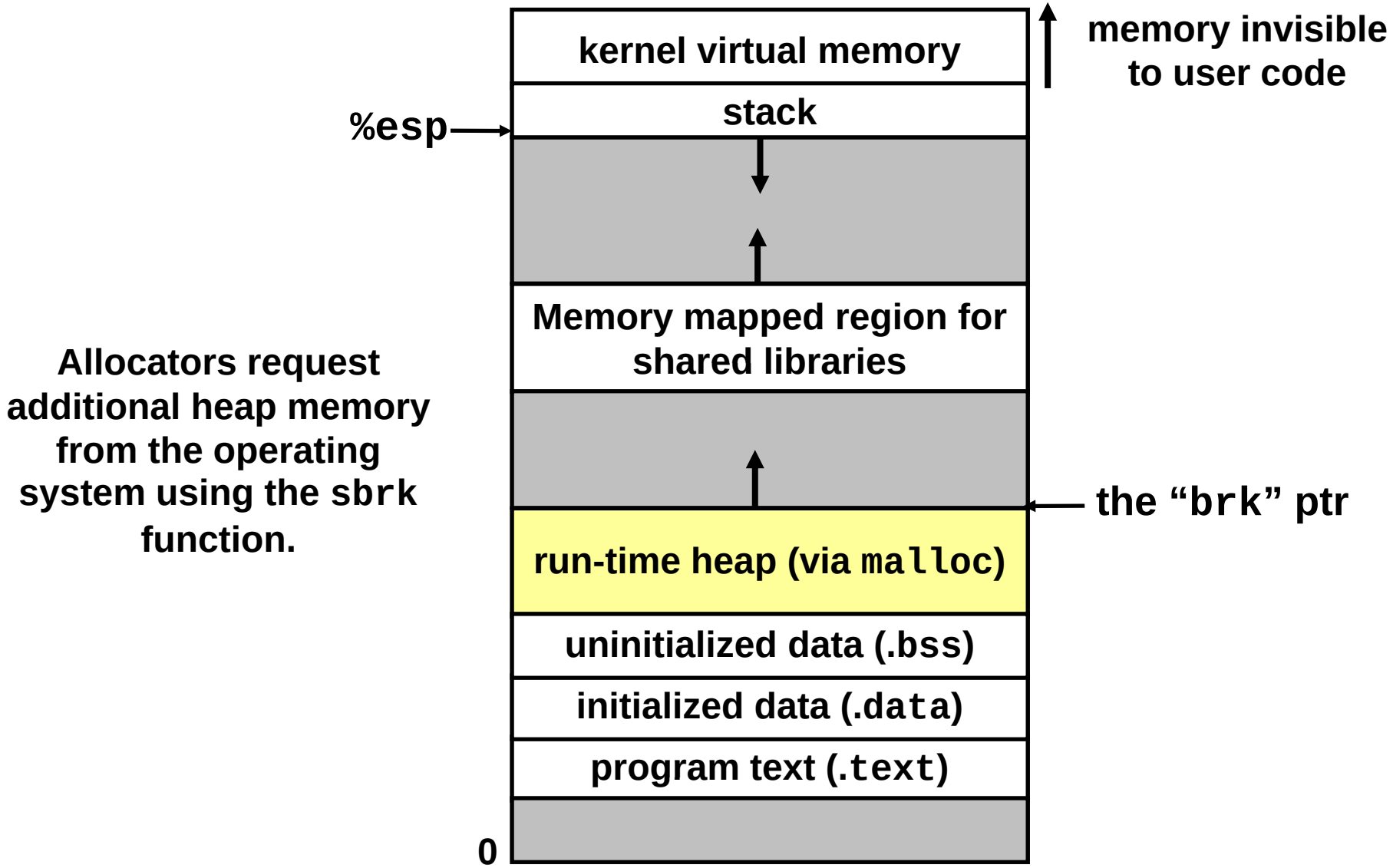
- 프로그램이 실행되기 전에는 크기를 알 수 없는 자료 구조를 위해 사용
- 예 : n 개의 문자를 화면에서 읽어 들여서 배열에 저장하고자 할 때, n 과 문자를 차례로 받아 들여서 실행하는 경우
  - 고정 크기의 배열로도 구현 가능
  - **int array[MAX_SIZE];**
  - 이와 같이 배열의 크기를 알 수 없을 때, 최대값으로 배열을 구현하는 것은 나쁜 생각
    - 시스템의 가능한 메모리 사용량을 알 수 없다
    - MAX_SIZE 보다 많은 입력을 원한다면?
    - MAX_SIZE 값을 계속 바꿔서 다시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

동적 메모리 할당



- 직접 vs 간접(Explicit vs. Implicit) Memory Allocator
  - 직접 할당 : **application allocates and frees space**
    - E.g., `malloc` and `free` in C
  - 간접 할당 : **application allocates, but does not free space**
    - E.g. garbage collection in Java, ML or Lisp
- Allocation
  - **In both cases the memory allocator provides an abstraction of memory as a set of blocks**
  - 응용프로그램에 **free** 메모리 블럭을 나눠준다
- Will discuss simple explicit memory allocation

프로세스의 메모리 이미지



Malloc 패키지

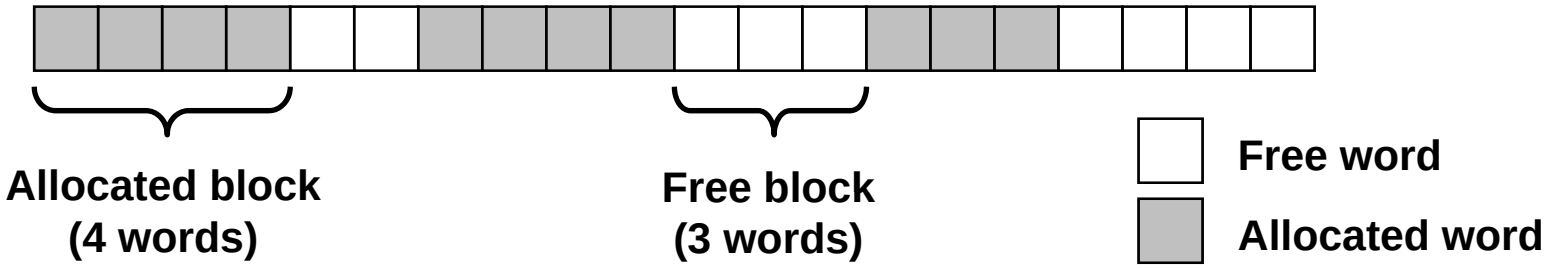
- `#include <stdlib.h>`
- `void *malloc(size_t size)`
  - **If successful:**
    - Returns a pointer to a memory block of at least `size` bytes, (typically) aligned to 8-byte boundary.
    - If `size == 0`, returns NULL
  - **If unsuccessful: returns NULL (0) and sets `errno`.**
- `void free(void *p)`
  - **Returns the block pointed at by `p` to pool of available memory**
  - **`p` must come from a previous call to `malloc` or `realloc`.**
- `void *realloc(void *p, size_t size)`
  - **Changes size of block `p` and returns pointer to new block.**
  - **Contents of new block unchanged up to min of old and new size.**

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

- Assumptions made in this lecture
  - **Memory is word addressed (each word can hold a pointer)**

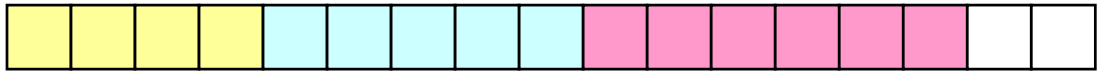


할당 예제

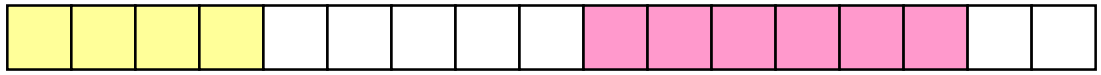
```
p1 = malloc(4)
```

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

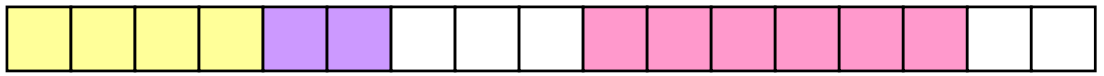
```
p3 = malloc(6)
```



```
free(p2)
```



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



제한사항

- 응용 프로그램
  - **Can issue arbitrary sequence of allocation and free requests**
  - **Free requests must correspond to an allocated block**
- 할당 프로그램
  - **Can't control number or size of allocated blocks**
  - **Must respond immediately to all allocation requests**
    - *i.e.*, can't reorder or buffer requests
  - **Must allocate blocks from free memory**
    - *i.e.*, can only place allocated blocks in free memory
  - **Must align blocks so they satisfy all alignment requirements**
    - 8 byte alignment for GNU malloc (`libc` malloc) on Linux boxes
  - **Can only manipulate and modify free memory**
  - **Can't move the allocated blocks once they are allocated**
    - *i.e.*, compaction is not allowed

우수한 malloc/free 프로그램의 목표

- Primary goals
  - **Good time performance for `malloc` and `free`**
    - Ideally should take constant time (not always possible)
    - Should certainly not take linear time in the number of blocks
  - **Good space utilization**
    - User allocated structures should be large fraction of the heap.
    - Want to minimize “fragmentation”.
- Some other goals
  - **Good locality properties**
    - Structures allocated close in time should be close in space
    - “Similar” objects should be allocated close in space
  - **Robust**
    - Can check that `free(p1)` is on a valid allocated object `p1`
    - Can check that memory references are to allocated space

성능 지표 : Throughput

- Given some sequence of `malloc` and `free` requests:
  - $R_0, R_1, ..., R_k, ..., R_{n-1}$
- Want to maximize throughput and peak memory utilization.
  - **These goals are often conflicting**
- Throughput:
  - **Number of completed requests per unit time**
  - **Example:**
    - 5,000 `malloc` calls and 5,000 `free` calls in 10 seconds
    - Throughput is 1,000 operations/second.
  - **alloc**과 **free** 함수의 평균 처리 시간을 최소화하면 **throughput**을 극대화 할 수 있다

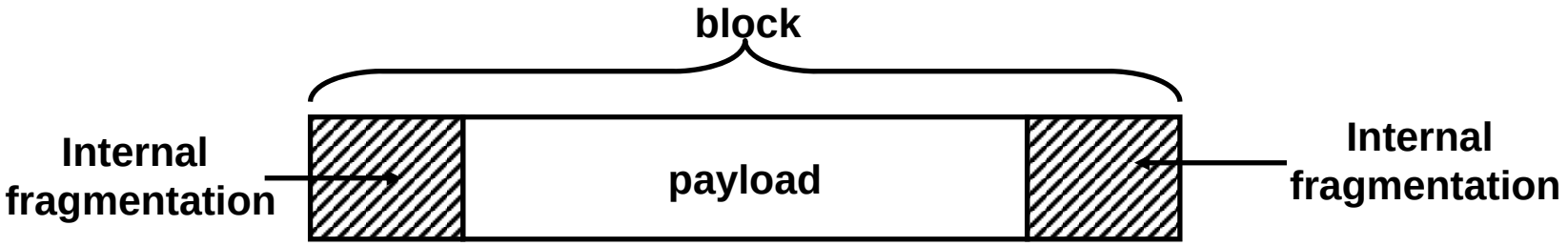
성능지표 : 순간 최대 메모리 사용율

- 좋은 프로그래머는 가상메모리의 크기도 제한되어 있다는 것을 알고 작업한다
  - 따라서 효율적으로 관리해야 한다
- 효율적인 heap 사용 지표 : peak heap utilization

- Given some sequence of `malloc` and `free` requests:
  - $R_0, R_1, ..., R_k, ..., R_{n-1}$
- *Def: Aggregate payload P_k:*
  - **`malloc(p)` results in a block with a *payload* of p bytes..**
  - **After request R_k has completed, the *aggregate payload* P_k is the sum of currently allocated payloads.**
- *Def: Current heap size is denoted by H_k*
  - **Assume that H_k is monotonically nondecreasing**
- *Def: Peak memory utilization:*
  - **After *k* requests, *peak memory utilization* is:**
    - $U_k = (max_{i<=k} P_i) \ / \ H_k$
- 구현 목표 : U_{n-1} 을 가능한 모든 작업 순서에 대해서 극대화한다

내부 메모리 단편화(Internal Fragmentation)

- Poor memory utilization caused by *fragmentation*.
  - **Comes in two forms: internal and external fragmentation**
- Internal fragmentation
  - **For some block, internal fragmentation is the difference between the block size and the payload size.**



  - **Caused by overhead of maintaining heap data structures, padding for alignment purposes, or explicit policy decisions (e.g., not to split the block).**
  - **Depends only on the pattern of *previous* requests, and thus is easy to measure.**

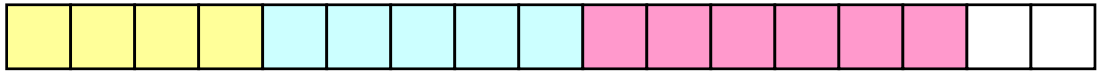
외부 메모리 단편화(External Fragmentation)

Occurs when there is enough aggregate heap memory, but no single free block is large enough

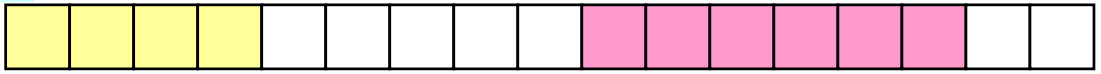
```
p1 = malloc(4)
```

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

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



```
free(p2)
```



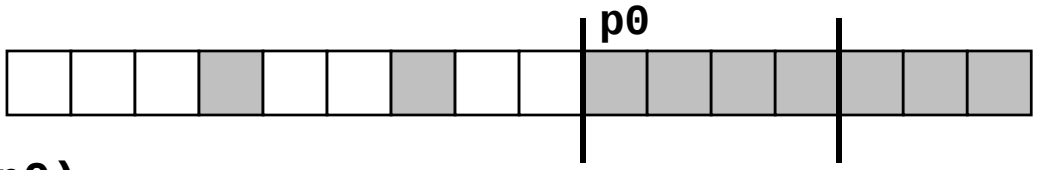
```
p4 = malloc(6)
```

oops!

External fragmentation depends on the pattern of *future* requests, and thus is difficult to measure.

구현시 고려할 점

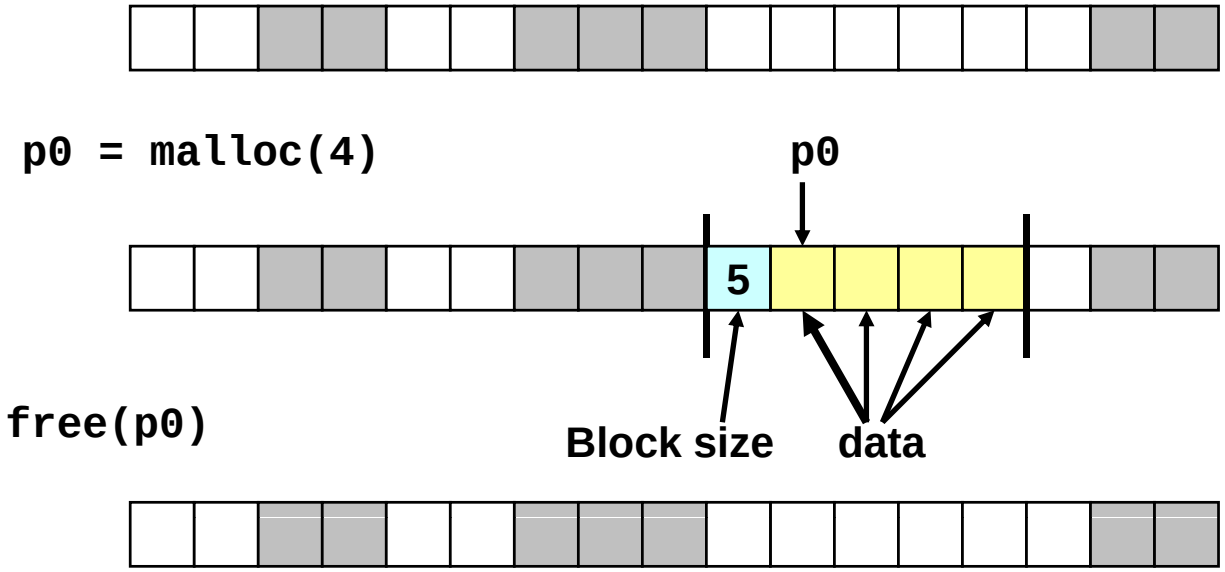
- How do we know how much memory to free just given a pointer?
- How do we keep track of the free blocks?
- What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
- How do we pick a block to use for allocation -- many might fit?
- How do we reinsert freed block?




```
free(p0)

 p1 = malloc(1)
```

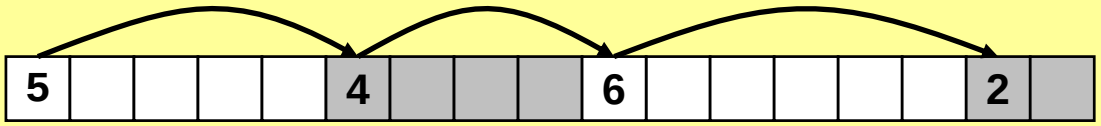
얼마만 큼 Free 시켜야 하는지 결정하기

- Standard method
  - **Keep the length of a block in the word preceding the block.**
    - This word is often called the *header field* or *header*
  - **Requires an extra word for every allocated block**

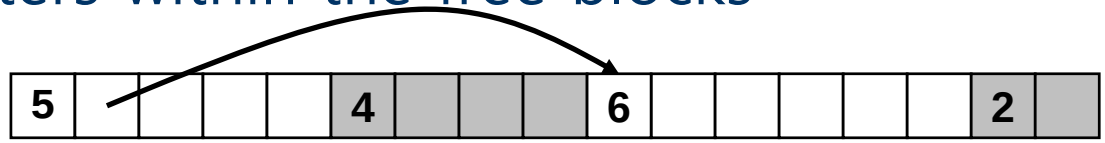


Free 블럭 관리하기

- *Method 1*: *Implicit list* using lengths -- links all blocks



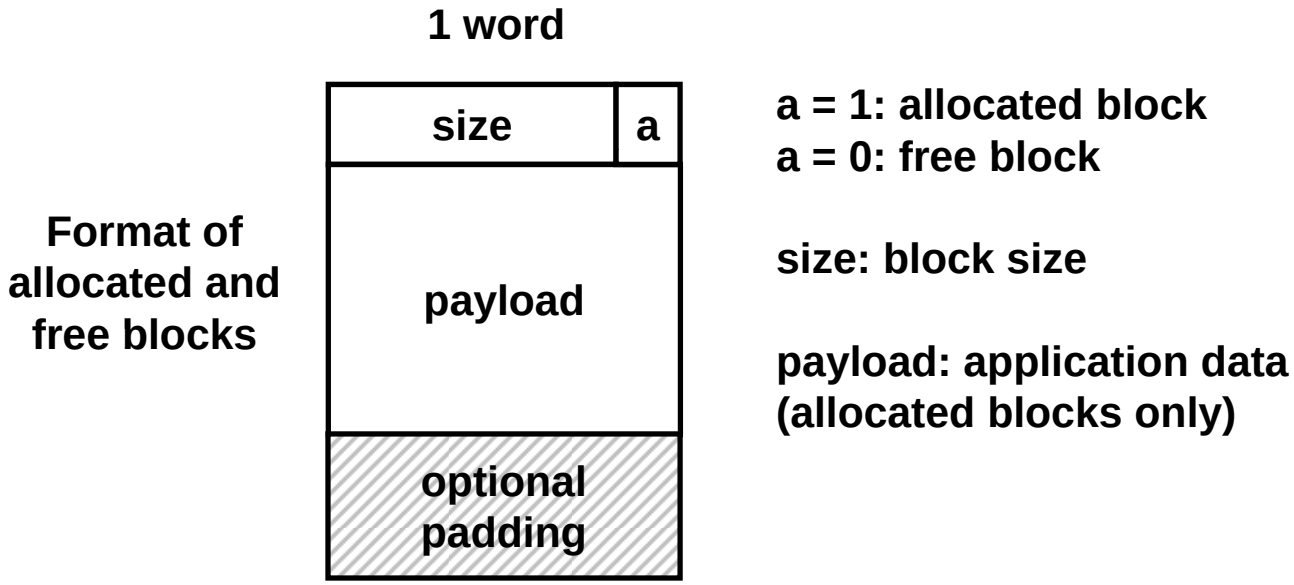
- *Method 2*: *Explicit list* among the free blocks using pointers within the free blocks



- *Method 3*: *Segregated free list*
  - Different free lists for different size classes
- *Method 4*: Blocks sorted by size
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

방법 1: 간접 리스트 방식(Implicit List)

- Need to identify whether each block is free or allocated
  - **Can use extra bit**
  - **Bit can be put in the same word as the size if block sizes are always multiples of two (mask out low order bit when reading size).**



간접 리스트 : Free 블록 찾기

- *First fit:*
  - Search list from beginning, choose first free block that fits

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

  - Can take linear time in total number of blocks (allocated and free)
  - In practice it can cause “splinters” at beginning of list
- *Next fit: (Donald Knuth 에 의해 제안됨)*
  - Like first-fit, but search list from location of end of previous search
  - Research suggests that fragmentation is worse
- *Best fit:*
  - Search the list, choose the free block with the closest size that fits
  - Keeps fragments small --- usually helps fragmentation
  - Will typically run slower than first-fit
- 세가지 검색 방법을 비교할 수 있는가?

비트 필드 용법

- How to represent the Header:
- Masks and bitwise operators

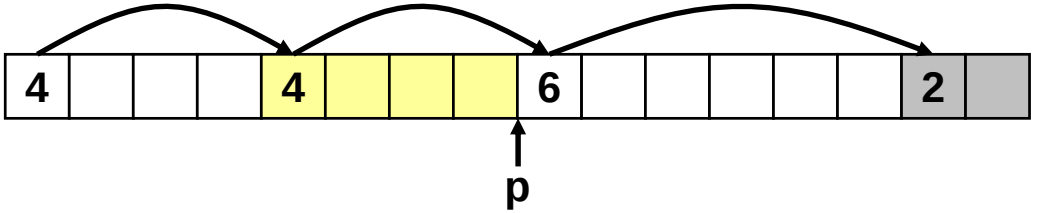
```
#define SIZEMASK              (~0x7)
#define PACK(size, alloc)  ((size) | (alloc))
#define getSize(x)           ((x)->size & SIZEMASK)
```

- bitfields

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

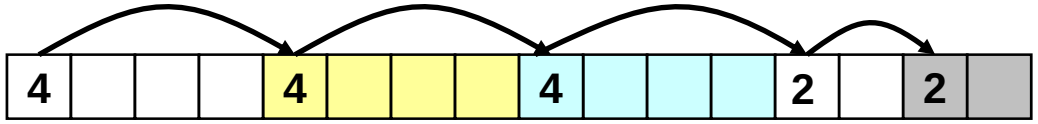
간접 리스트 : Free 블록 할당

- Allocating in a free block - *splitting*
  - **Since allocated space might be smaller than free space, we might want to split the block**




```
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)

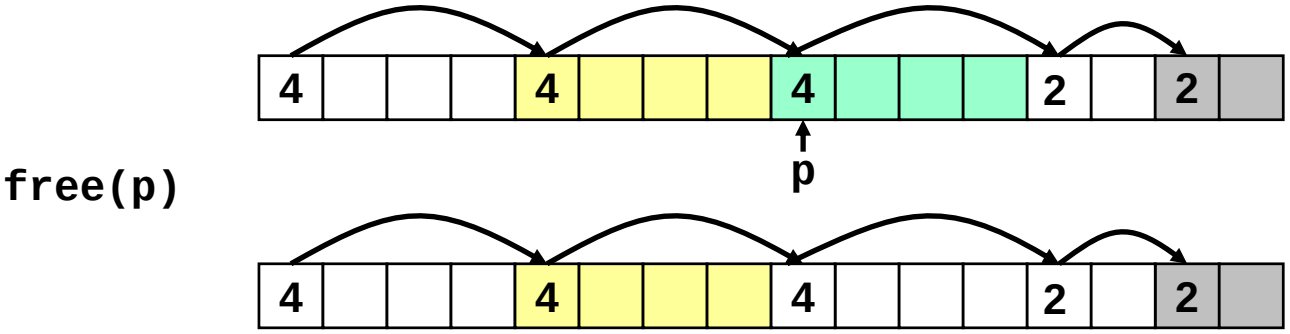


간접 리스트 : 블록 free 시키기

- Simplest implementation:
  - **Only need to clear allocated flag**

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

  - **But can lead to "false fragmentation"**

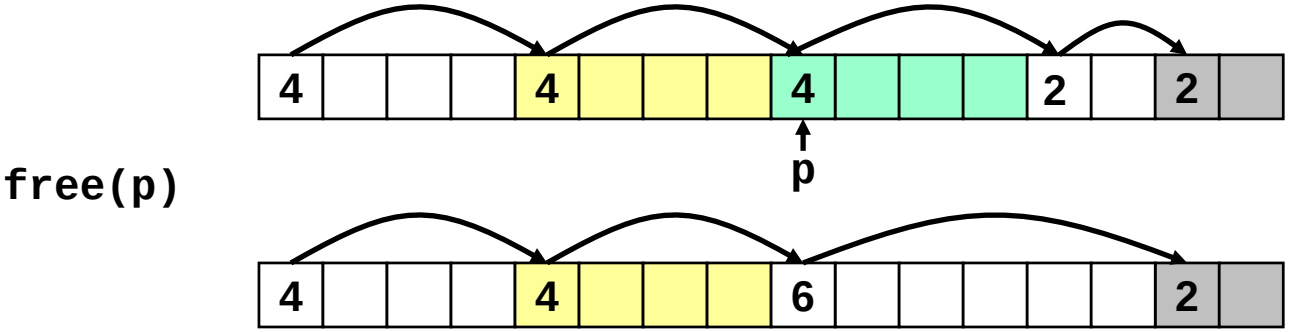


There is enough free space, but the allocator won't be able to find it

간접 리스트 : 연결(Coalescing)

- 다음 또는 이전 블록이 free 하면 함께 연결해서 더 큰 free 블럭을 만든다
  - **Coalescing with next block**

```
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
```



  - **But how do we coalesce with previous block?**

간접 리스트 : 양방향 연결

- *Boundary tags* [Knuth73]
  - Replicate size/allocated word at bottom of free blocks
  - Allows us to traverse the "list" backwards, but requires extra space
  - Important and general technique!