Memory Management

CMPU 334 – Operating Systems
Jason Waterman

Memory API: `malloc()`

```
#include <stdlib.h>

void* malloc(size_t size)
```

- Allocate a memory region on the heap
  - Argument
    - `size_t size` : size of the memory block (in bytes)
    - `size_t` is an unsigned integer type capable of holding the size of any valid memory request
    - Often used with `sizeof(type)` operator which returns the size of the argument
  - Return
    - Success: a void type pointer to the memory block allocated by `malloc`
    - Fail: a `null` pointer

Memory API: `sizeof()`

- Good coding style to use `sizeof` in `malloc` instead of requesting the number of bytes directly
- Two types of results of `sizeof` with variables
  - The actual size of 'x' is known at run-time

```
int *x = malloc(10 * sizeof(int));
printf("%d\n", sizeof(x));
```

```
8
```

  - The actual size of 'x' is known at compile-time

```
int x[10];
printf("%d\n", sizeof(x));
```

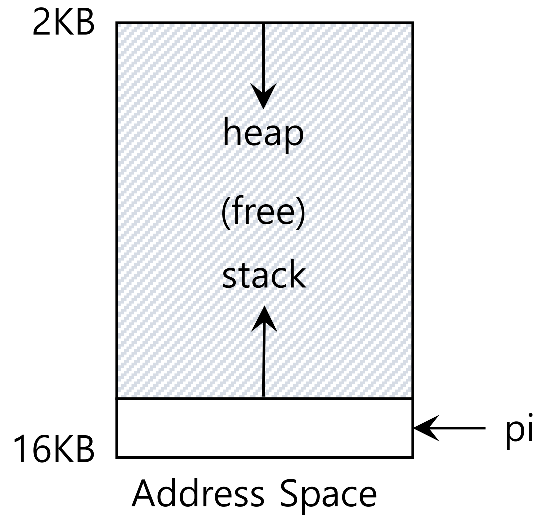
```
40
```

Memory API: `free()`

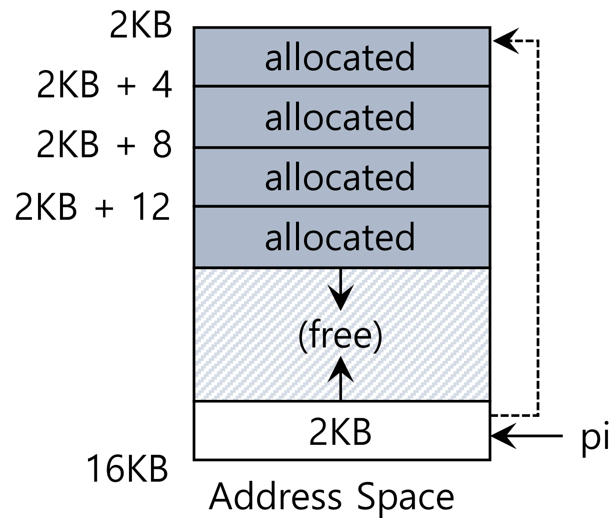
```
#include <stdlib.h>

void free(void* ptr)
```

- Free a memory region allocated by a call to `malloc`
  - Argument
    - `void *ptr` : a pointer to a memory block allocated with `malloc`
  - Returns
    - none

Memory Allocating



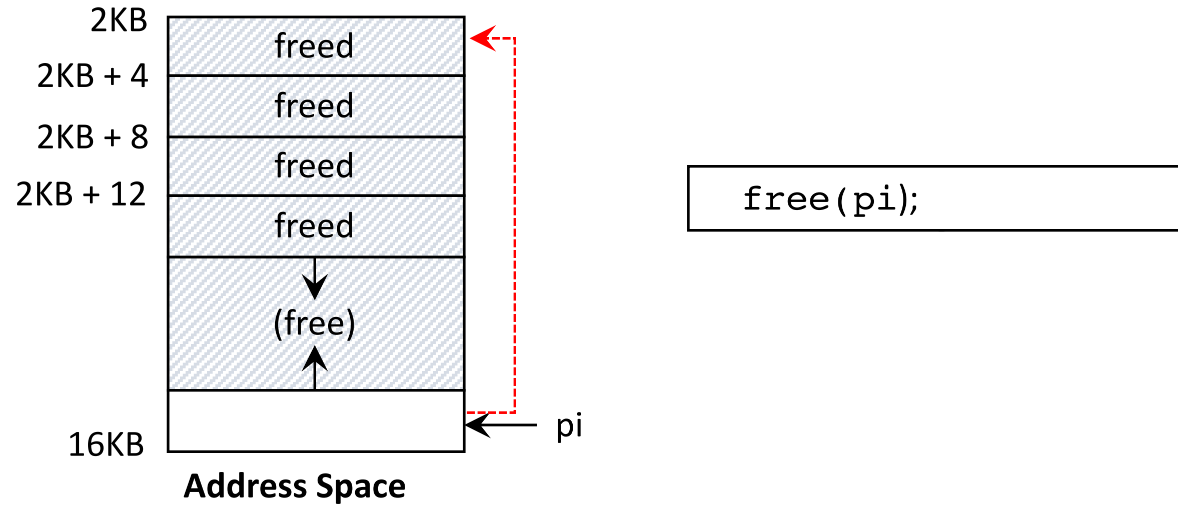

```
int *pi; // local variable
```

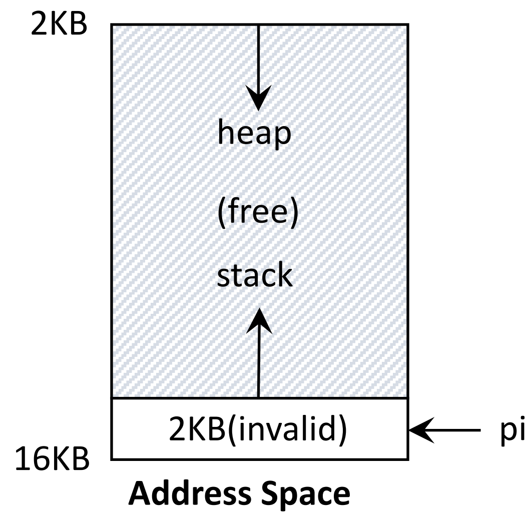


-------------> points to

```
pi = (int *)malloc(sizeof(int)* 4);
```

Memory Freeing

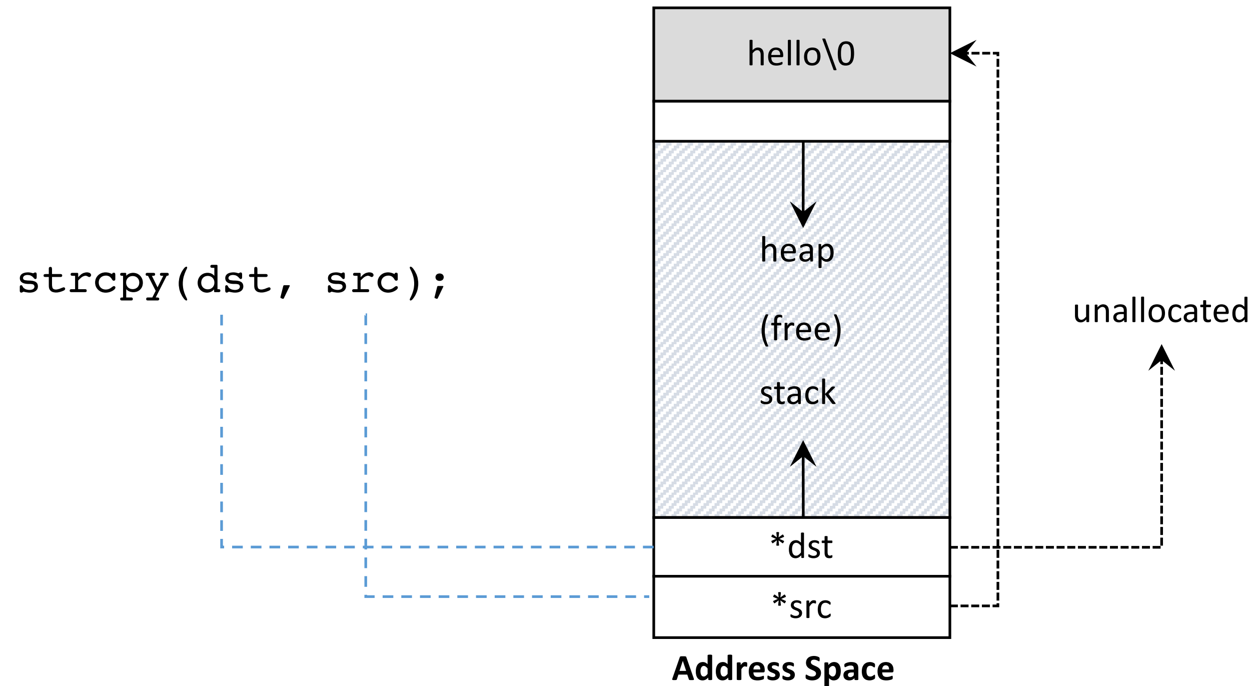




Forgetting To Allocate Memory

- Incorrect code

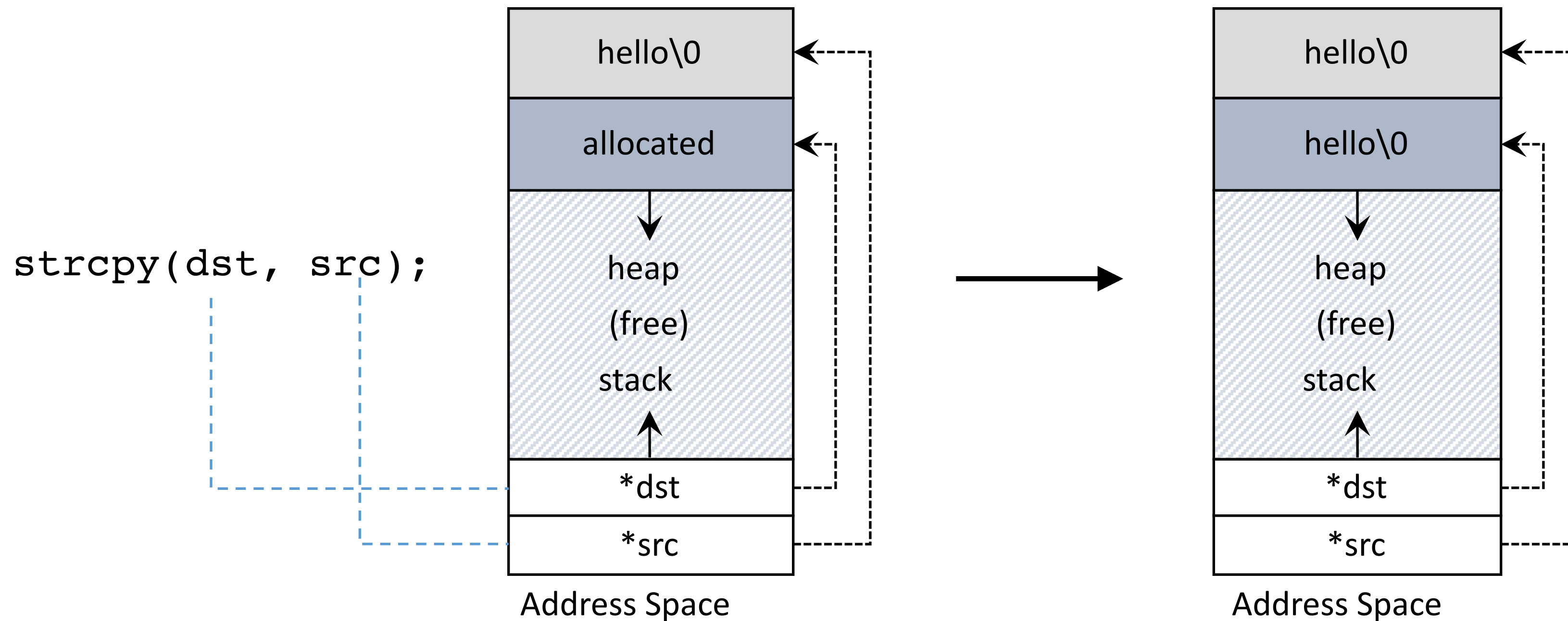
```
char *src = “hello”; //character string constant
char *dst;           //unallocated
strcpy(dst, src);    //segfault and die
```



Forgetting To Allocate Memory (Cont.)

- Correct code

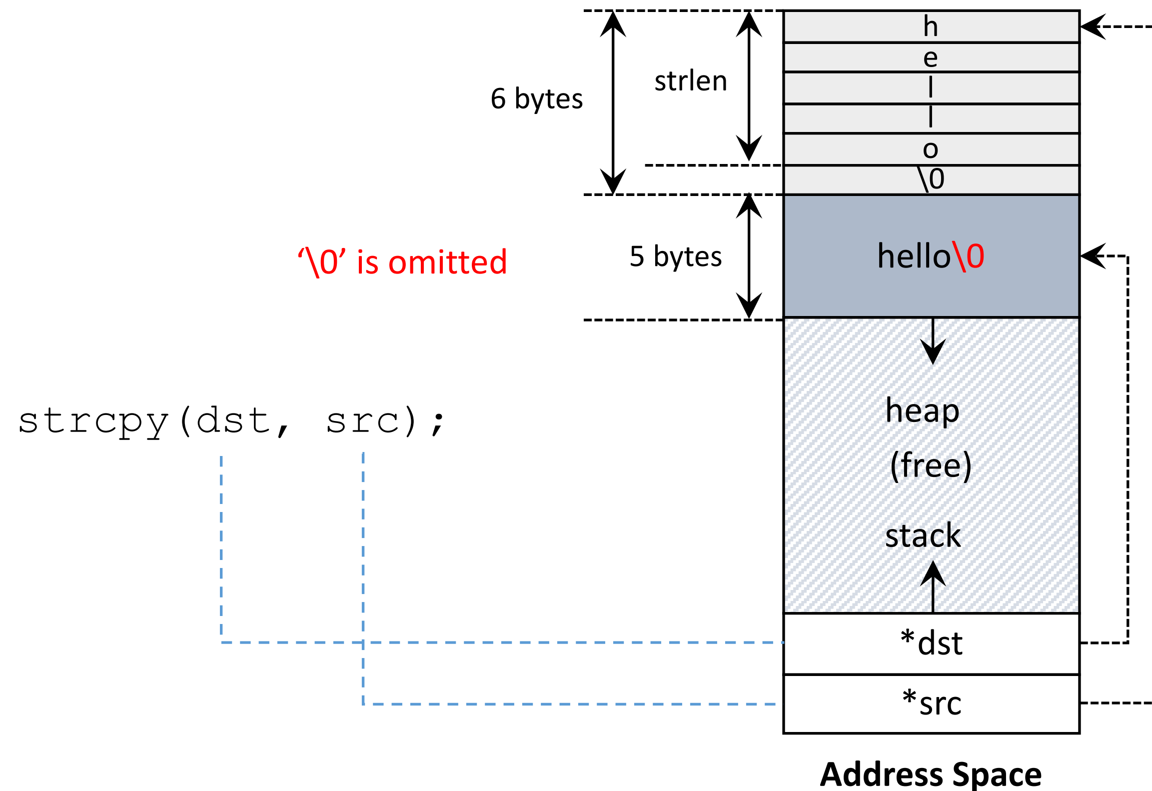
```
char *src = "hello";   //character string constant
char *dst (char *)malloc(strlen(src) + 1 ); // allocated
strcpy(dst, src);      //work properly
```



Not Allocating Enough Memory

- Incorrect code, but “works” properly

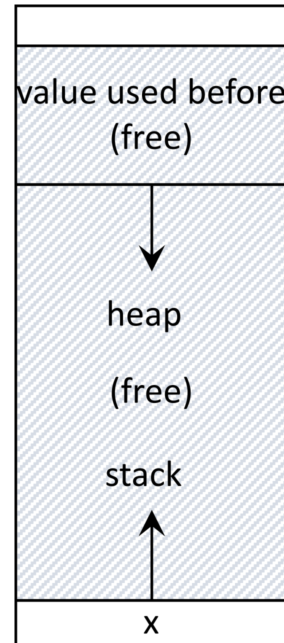
```
char *src = "hello"; //character string constant
char *dst (char *)malloc(strlen(src)); // too small
strcpy(dst, src);    //works "properly"
```



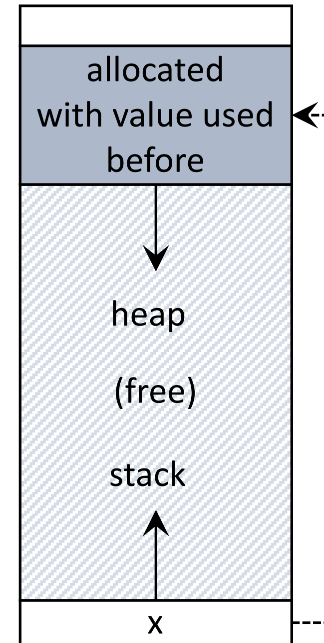
Forgetting to Initialize

- Encounter an uninitialized read

```
int *x = (int *)malloc(sizeof(int)); // allocated
printf("*x = %d\n", *x); // uninitialized memory access
```



Address Space



Address Space

Memory Leak

- A program runs out of memory and eventually dies

unused : unused, but not freed

Address Space (1)	Address Space (2)	Address Space (3)
allocated	unused	unused
heap (free) stack	allocated	unused
*a	heap (free) stack	unused
	*b	allocated
	*a	(free)
		*d
		*c
		*b
		*a

run out of memory

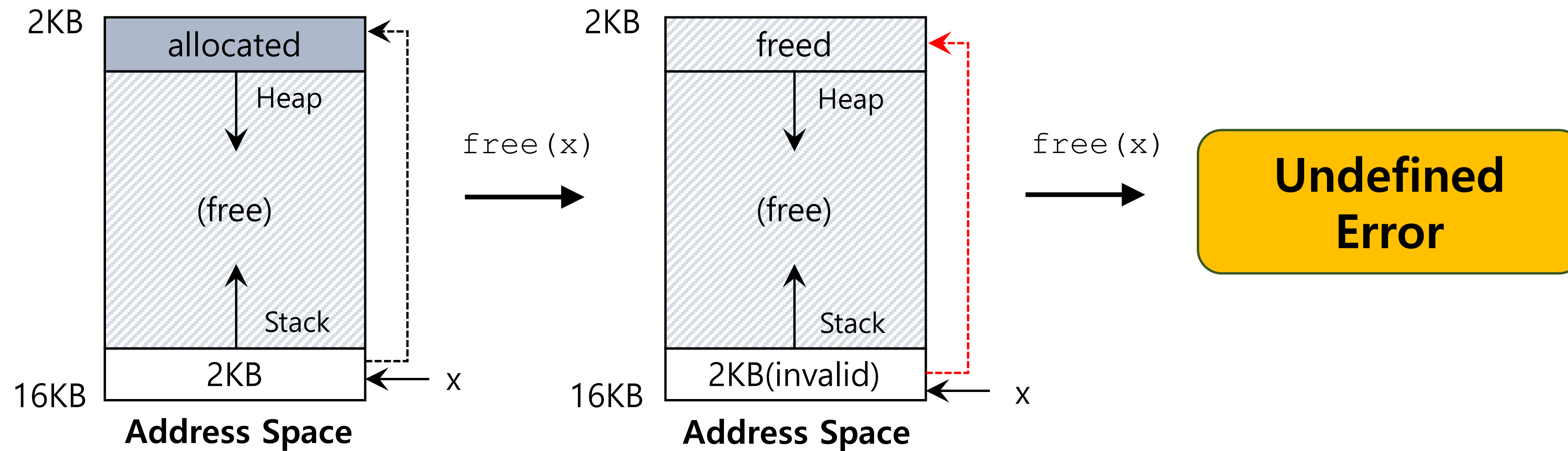
Dangling Pointer

- Freeing memory before it is finished being used

Double Free

- Free memory that was freed already

```
int *x = (int *)malloc(sizeof(int)); // allocated
free(x); // free memory once, OK
free(x); // free repeatedly, bad
```



Other Memory APIs: `calloc()`

```
#include <stdlib.h>

void *calloc(size_t nmemb, size_t size)
```

- Allocate memory on the heap and zeroes it before returning
  - Arguments
    - `nmemb`: number of elements to allocate
    - `size`: size of one element
  - Returns
    - Success: a void type pointer to the memory block allocated by `calloc`
    - Fail: a null pointer

Other Memory APIs: realloc()

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size)
```

- Change the size of memory block
  - A pointer returned by `realloc` may be either the same as `ptr` or a new
  - Argument
    - `void *ptr`: Pointer to memory block allocated with `malloc`, `calloc`, or `realloc`
    - `size_t size`: New size for the memory block(in bytes)
  - Return
    - Success: Void type pointer to the memory block
    - Fail: Null pointer

System Calls

```
#include <unistd.h>

int brk(void *addr)
void *sbrk(intptr_t increment);
```

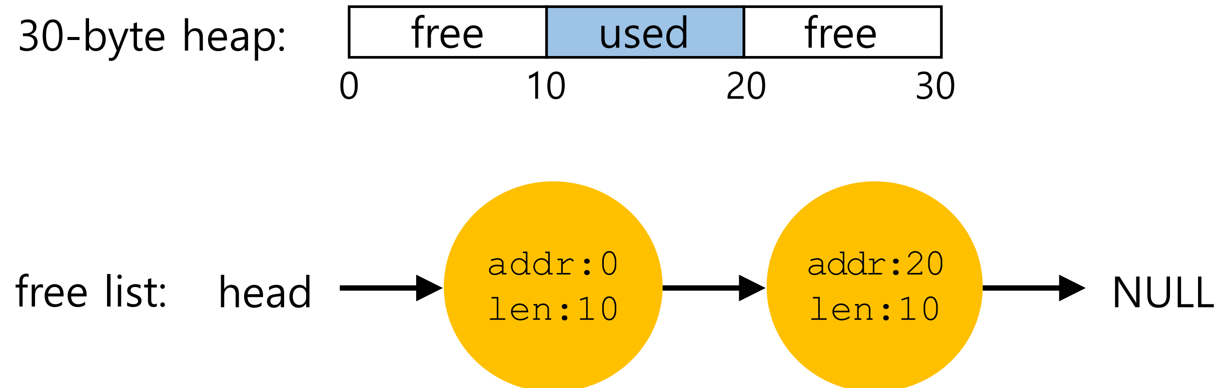
- `malloc` library call uses `brk` system call
  - `brk` is called to expand the program's *break*
    - *break*: The addressof **the end of the heap** in address space
  - `sbrk` increases the break by increment
  - Programmers **should never directly call** either `brk` or `sbrk`

Free space management

- How should free space be managed?
  - Variable-sized requests
- How can we minimize fragmentation
- Time and space requirements for allocation algorithms

Splitting

- Finding a free chunk of memory that can satisfy the request and splitting it into two
  - When request for memory allocation is **smaller** than the size of free chunks



Splitting(Cont.)

- Two 10-bytes free segment with **1-byte request**

30-byte heap:

free	used	free

0 10 20 30

free list: head → addr:0 len:10 → addr:20 len:10 → NULL

splitting 10 – byte free segment

30-byte heap:

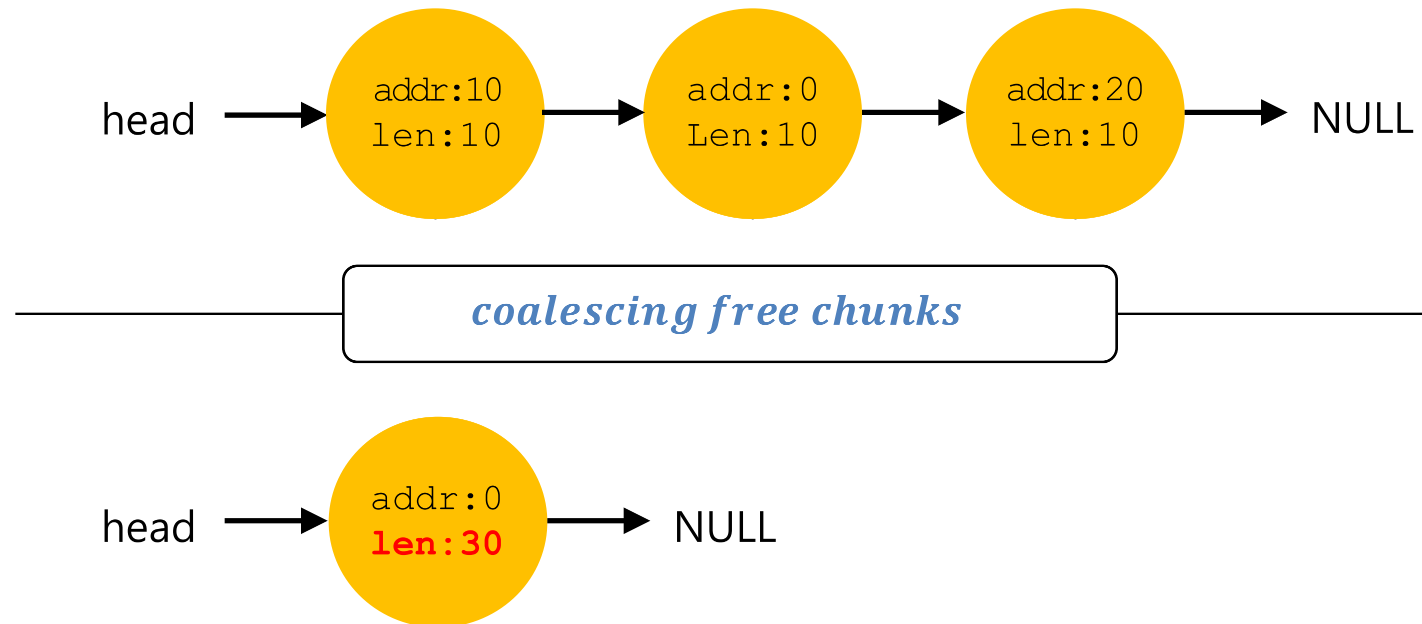
free	used		free

0 10 20 21 30

free list: head → addr:0 len:10 → **addr:21** len:9 → NULL

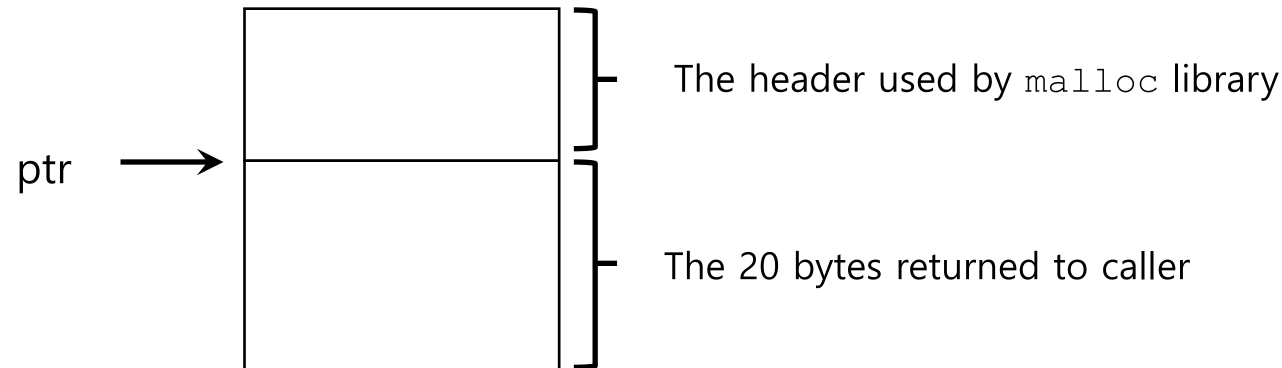
Coalescing

- If a user requests memory that is **bigger than the free chunk size**, the list will **not find** such a free chunk
- Coalescing: **Merge** returning a free chunk with existing chunks into a large single free chunk if **addresses** of them are **nearby**



Tracking The Size of Allocated Regions

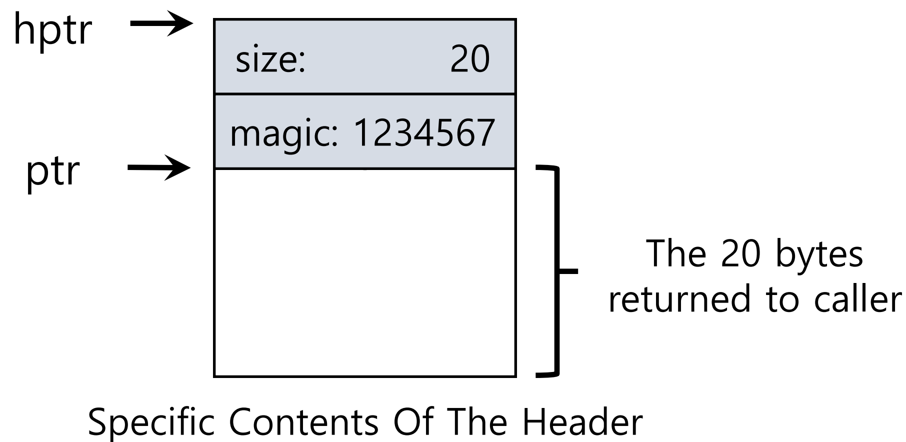
- The interface to `free(void *ptr)` does **not take** a **size parameter**
  - How does the library **know the size** of memory region that will be back **into free list**?
- Most allocators store **extra information** in a **header** block



An Allocated Region Plus Header

The Header of Allocated Memory Chunk

- The header minimally **contains the size** of the allocated memory region
- The header may also contain
  - Additional pointers to speed up deallocation
  - A magic number for integrity checking



Specific Contents Of The Header

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

A Simple Header

The Header of Allocated Memory Chunk(Cont.)

- The size for free region is the size of the header plus the size of the space allocated to the user
  - If a user request N bytes, the library searches for a free chunk of size N plus the size of the header
- Simple pointer arithmetic to find the header pointer

```
void free(void *ptr) {
        header_t *hptr = (char *)ptr - sizeof(header_t);
}
```

Embedding a Free List

- The memory-allocation library **initializes** the heap and **puts** the first element of **the free list** in the **free space**
  - The library **can't use** `malloc()` to build a list **within itself**
- Description of a node in the list

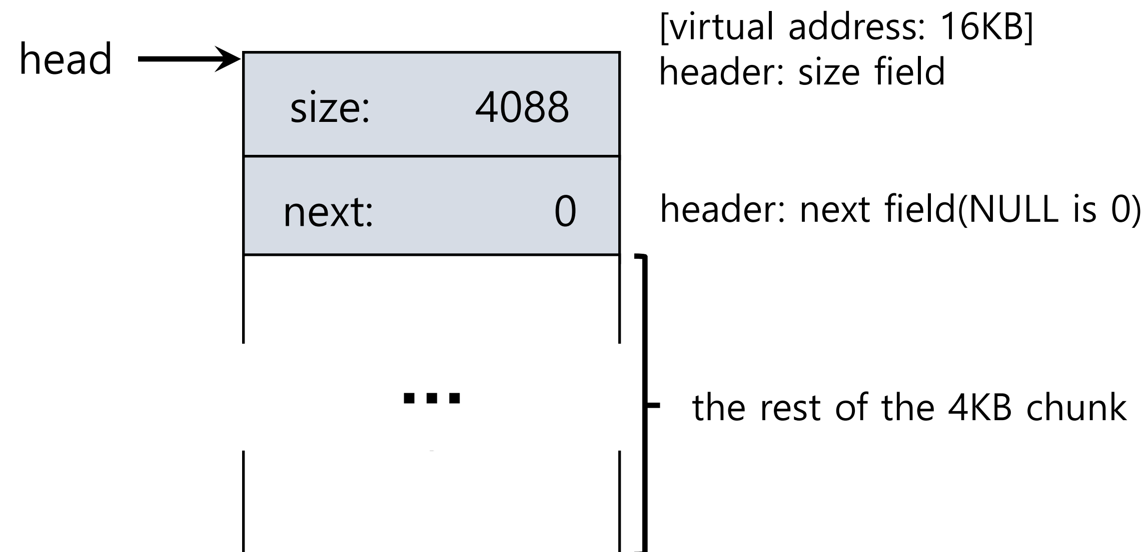
```
typedef struct __node_t {
        int size;
        struct __node_t *next;
} node_t;
```

- Building the heap and creating a free list
  - Assume the heap is built with **`mmap()`** system call

```
// mmap() returns a pointer to a chunk of free space
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;
```

A Heap With One Free Chunk

```
// mmap() returns a pointer to a chunk of free space
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;
```



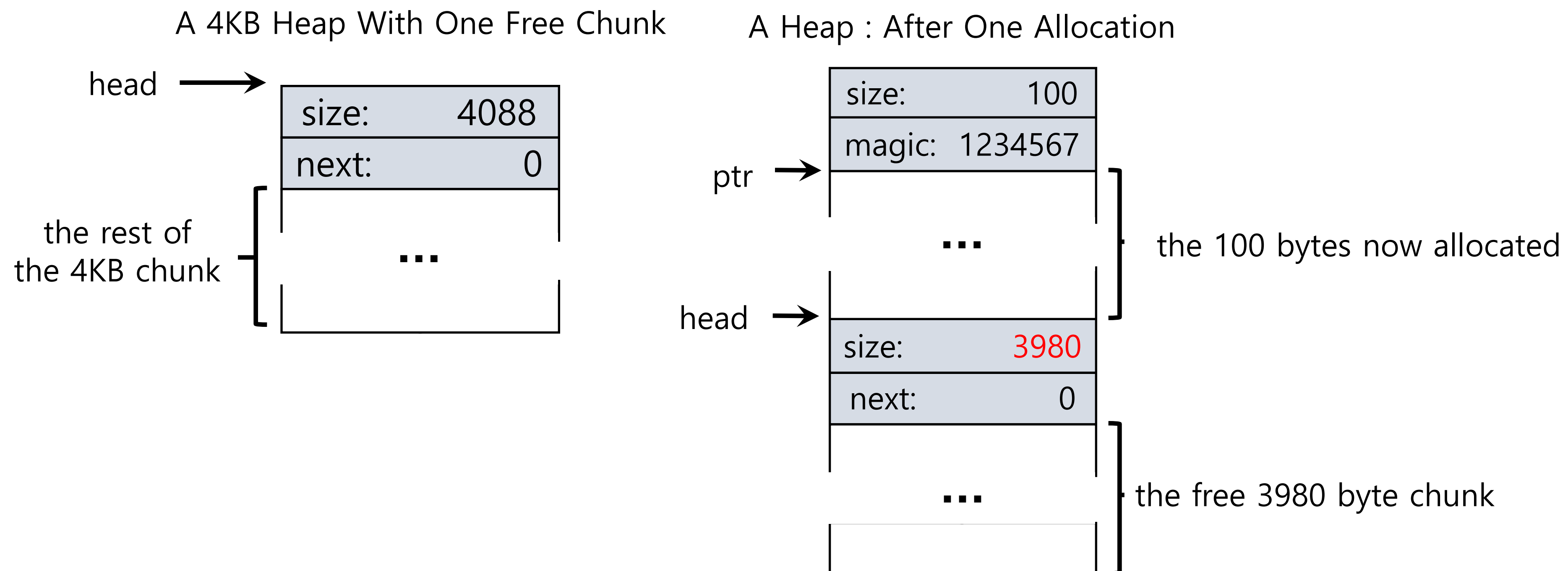
Embedding A Free List: Allocation

- If a chunk of memory is requested, the library **will first find** a chunk that is **large enough** to accommodate the request

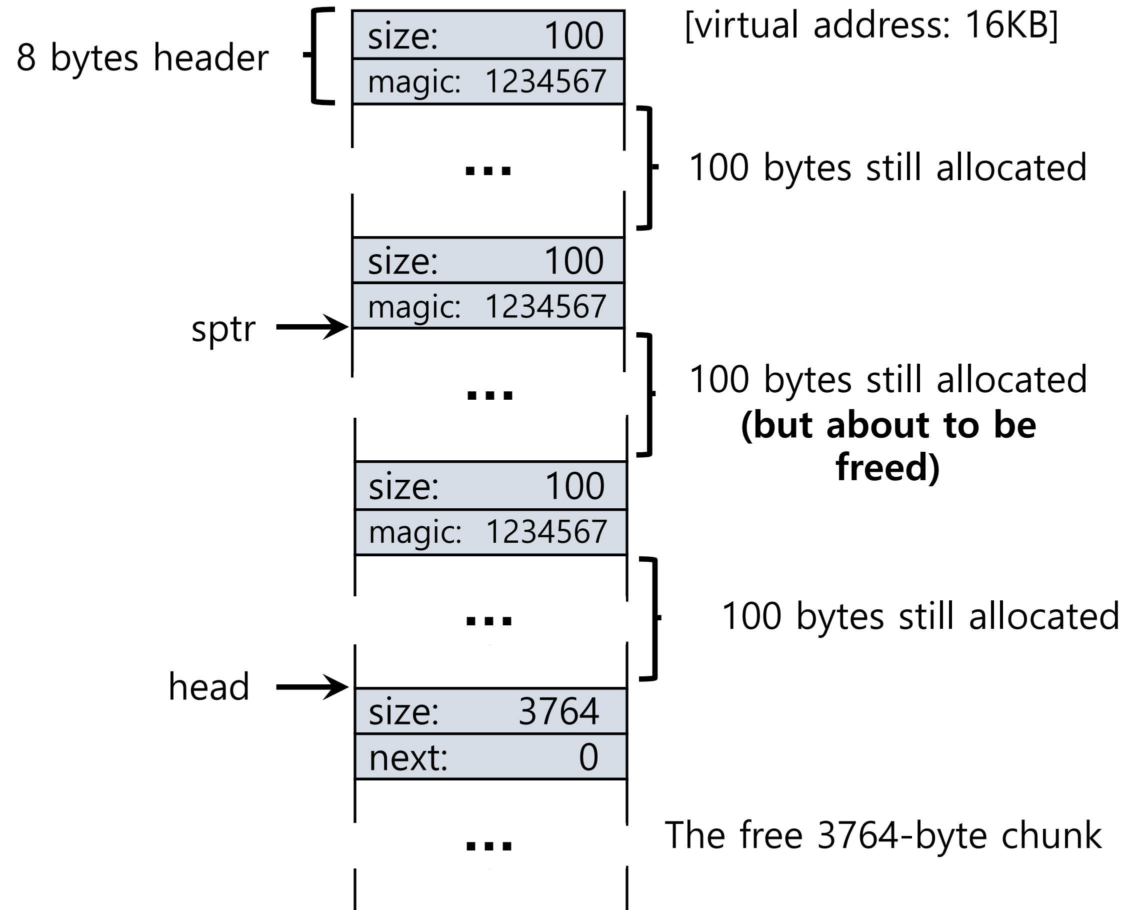
- The library will
  - **Split** the large free chunk into two
    - **One** for the **request** and the **remaining** free chunk
  - **Shrink** the size of free chunk in the list

Embedding A Free List: Allocation(Cont.)

- Example: a request for 100 bytes by `ptr = malloc(100)`
  - Allocating 108 bytes out of the existing one free chunk
  - shrinking the one free chunk to 3980(4088 minus 108)



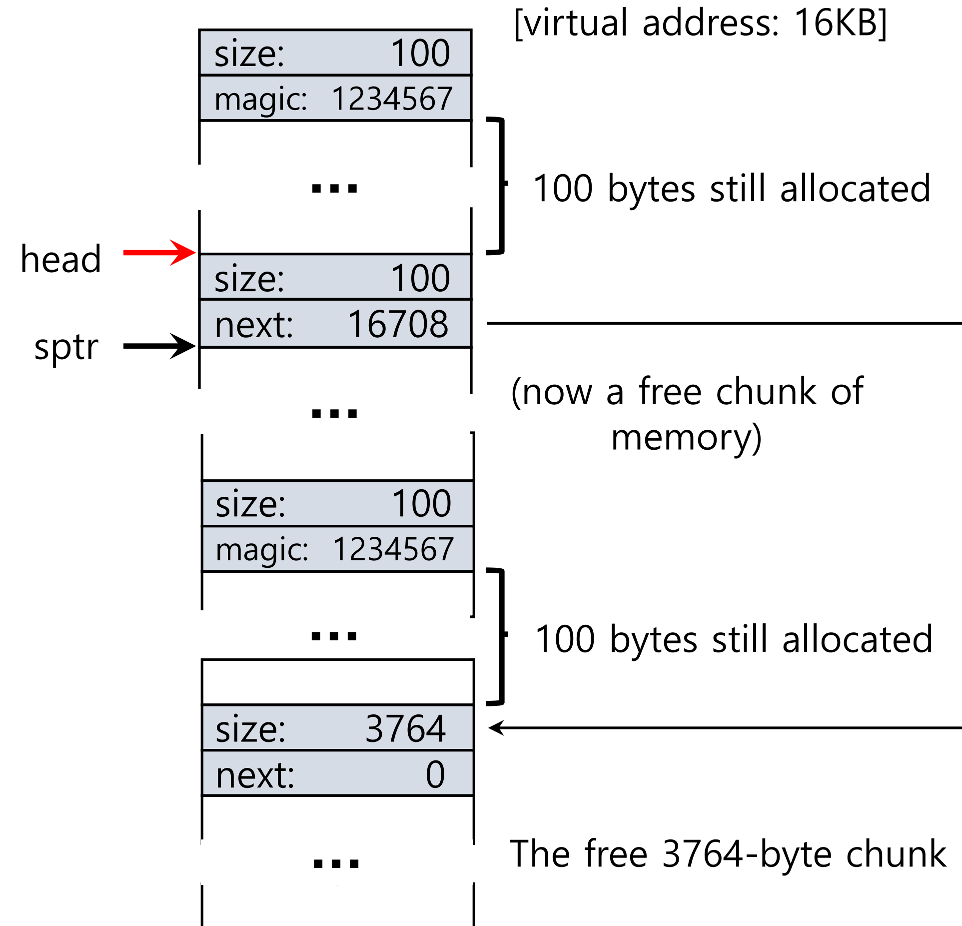
Free Space With Chunks Allocated



Free Space With Three Chunks Allocated

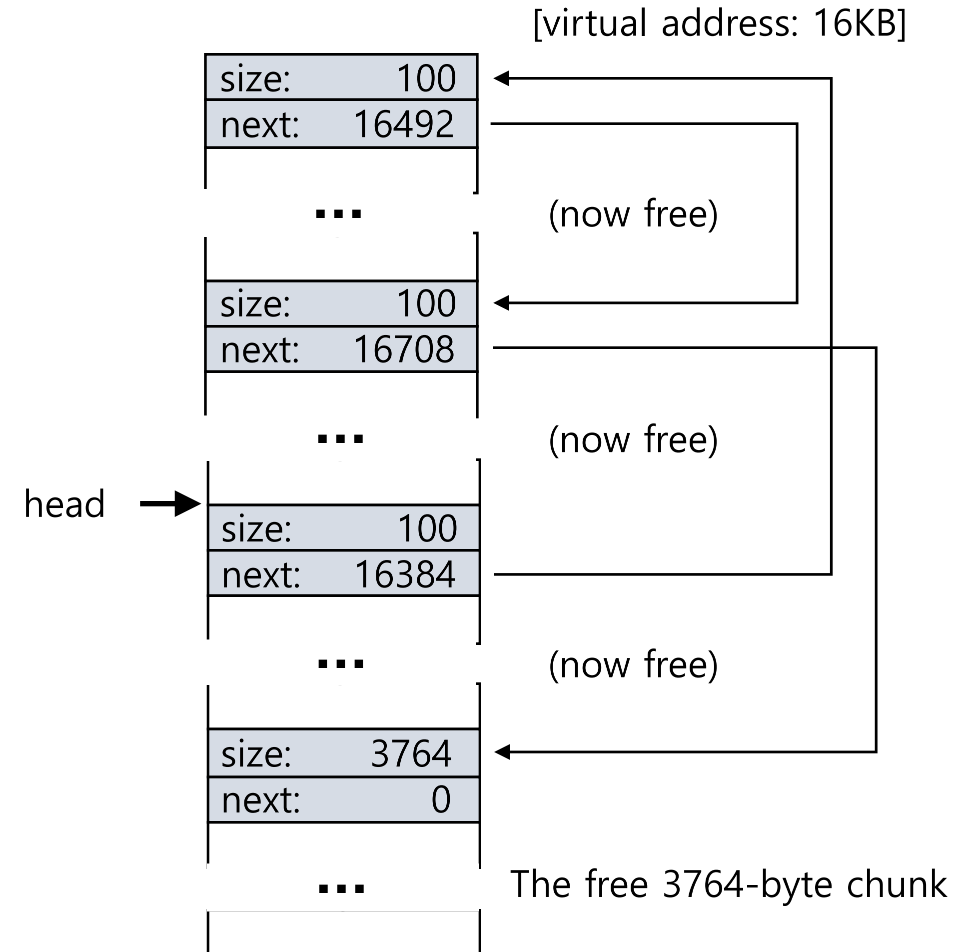
Free Space With `free()`

- Example: `free(sptr)`
  - The 100 byte chunk is **put back into** the free list
  - sptr->next == old head
  - It is now the head of the free list
  - The free list will **start** with **the small chunk**



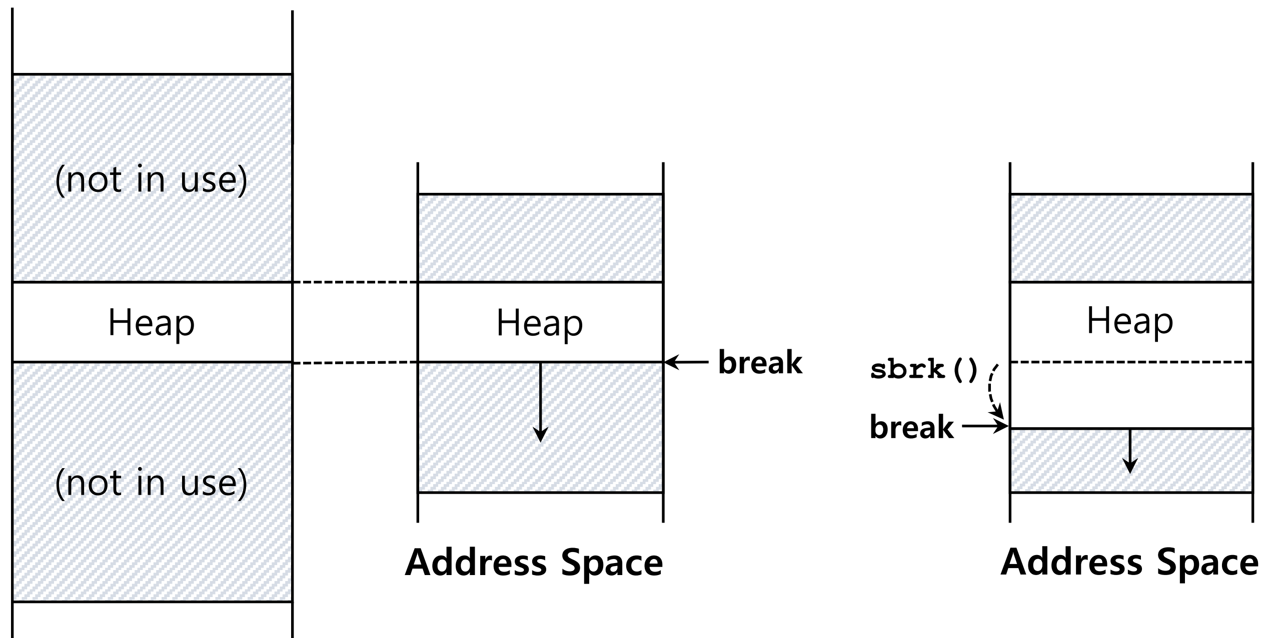
Free Space With Freed Chunks

- Let's assume that the last two in-use chunks are freed
- **External Fragmentation** occurs
  - **Coalescing** is needed in the list



Growing The Heap

- Most allocators **start** with **a small-sized heap** and then **request more** memory from the OS when they run out
  - e.g., `sbrk()`, `brk()` in most UNIX systems

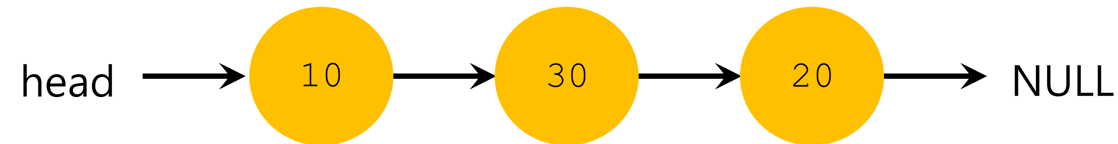


Managing Free Space: Basic Strategies

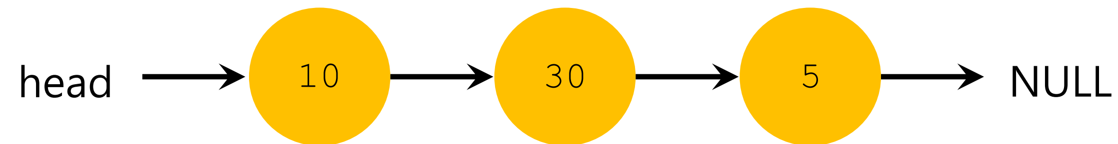
- Best Fit:
  - Finding free chunks that are **big or bigger than the request**
  - Returning the **one of smallest** in the chunks **in the group** of candidates
- Worst Fit:
  - Finding the **largest free chunks** and allocation the amount of the request
  - **Keeping the remaining chunk** on the free list
- First Fit:
  - Finding the **first chunk** that is **big enough** for the request
  - Returning the requested amount and remaining the rest of the chunk
- Next Fit:
  - Finding the first chunk that is big enough for the request
  - Searching at **where one was looking** at instead of the beginning of the list

Examples of Basic Strategies

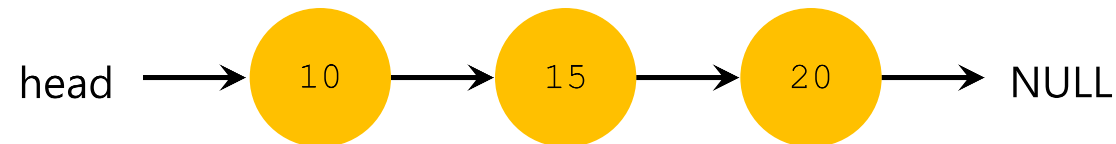
- Allocation Request Size 15



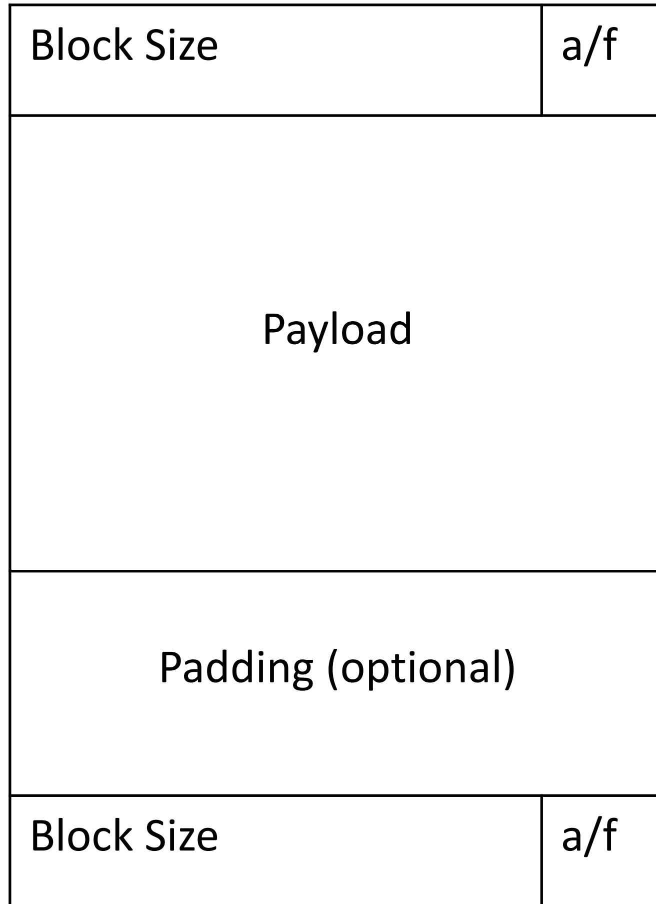
- Result of Best-fit



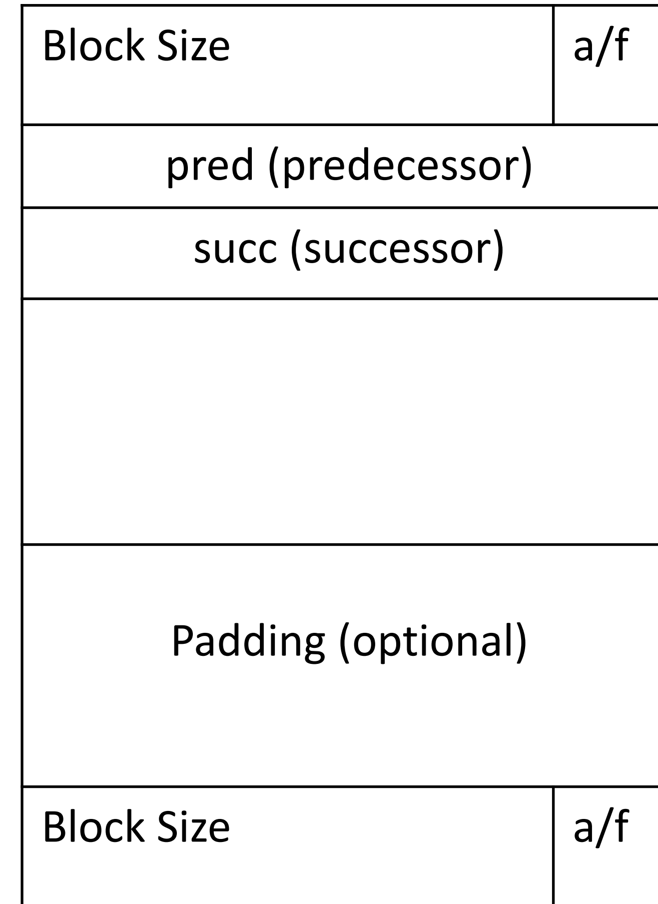
- Result of Worst-fit



Explicit Free List



Allocated Block



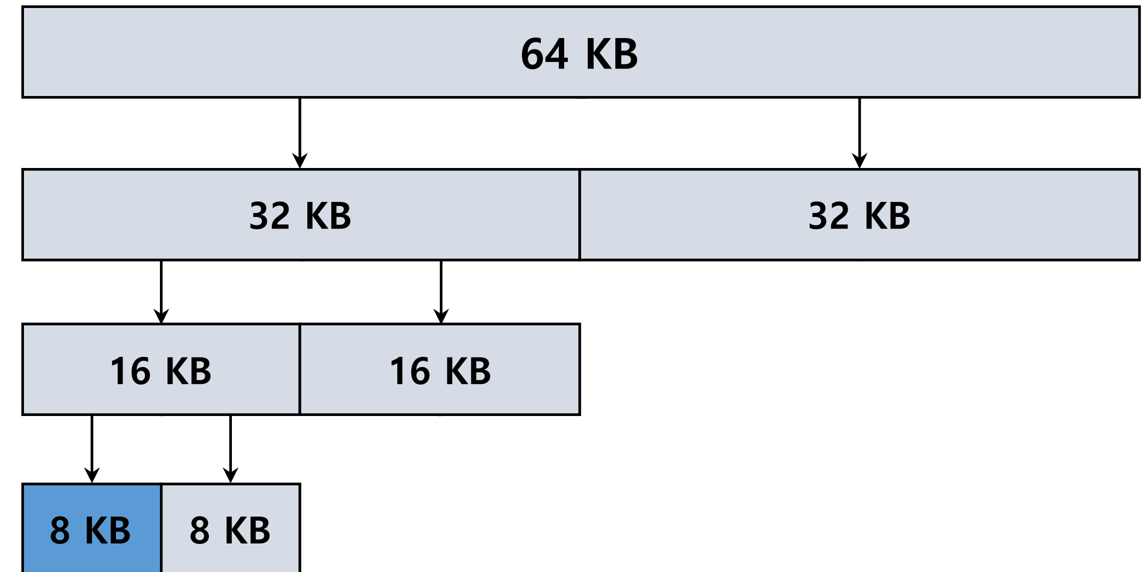
Free Block

Other Approaches

- Segregated List
  - Keeping free chunks in different size in a separate list for the size of popular request
  - New Complication:
    - **How much** memory should dedicate to **the pool of memory** that serves **specialized requests** of a given size?
  - **Slab allocator** handles this issue
- Slab Allocator
  - Allocate a number of object caches
    - The objects are likely to be requested frequently
    - e.g., locks, file-system inodes, etc.
    - Keeps freed objects in a particular list in their initialized state
  - **Request some memory** from a more general memory allocator when **a given cache is running low** on free space

Other Approaches

- Binary Buddy Allocation
  - The allocator **divides free space** by two **until a block** that is big enough to accommodate the request is **found**
- Buddy allocation can suffer from **internal fragmentation**
- Buddy system makes **coalescing** simple
  - **Coalescing** two blocks in to the next level of block



64KB free space for 7KB request