

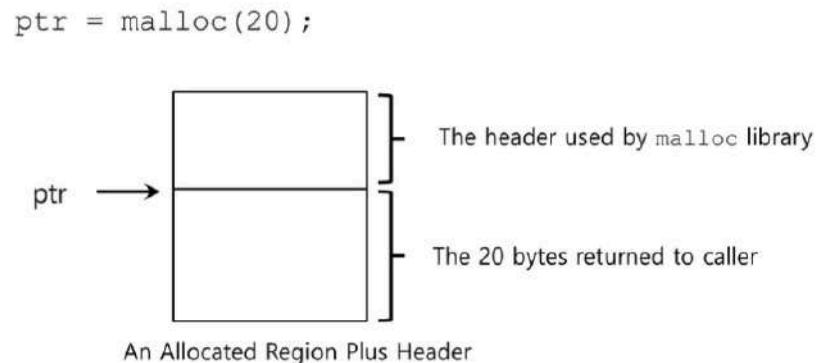
Recall that the interface to `free(void *ptr)` does not take a size parameter.

Then, how does the library know the size of the memory region that will be back into free list?

↳ most allocators store extra information in a header block.

\* The size for free region is the size of the header + size of the space allocated to the user.

i.e., if a user request N bytes, the library searches for a free chunk of size N + the size of the header.



What are the contents of the header?

- \* size of the allocated region
- \* "magic number", for additional integrity checking,  
etc..

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

What exactly happens when user calls free(ptr) ?

- \* The library uses the following to figure out where the header begins?

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

- \* library then determine whether the magic number matches the expected value as a sanity check.  
magic number mismatch → error reported!

- \* calculate the total size of the newly-freed region.  
size of region + size of the header.

## NOTE:

How magic number helps in debugging?

\* Detecting double frees

free(\*ptr)



→ magic number matches and gets updated

free (\*ptr)

→ allocator sees a different magic number and reports

Also can be used to detect,

double free

\* invalid frees

\* detecting memory corruption

\* Buffer overflows, etc.-

## Embedding a free list

In a typical list → for allocating a new node → call malloc().

but what about free list? → need to build it inside the free space itself.

\* Description of a node of the list:

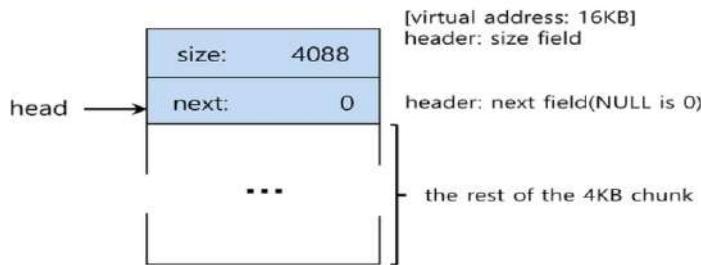
```
typedef struct __node_t {  
    int size;  
    struct __node_t *next;  
} nodet_t;
```

\* The process starts with some initial heap space from the kernel.

- assume that the heap is built via an 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;
```

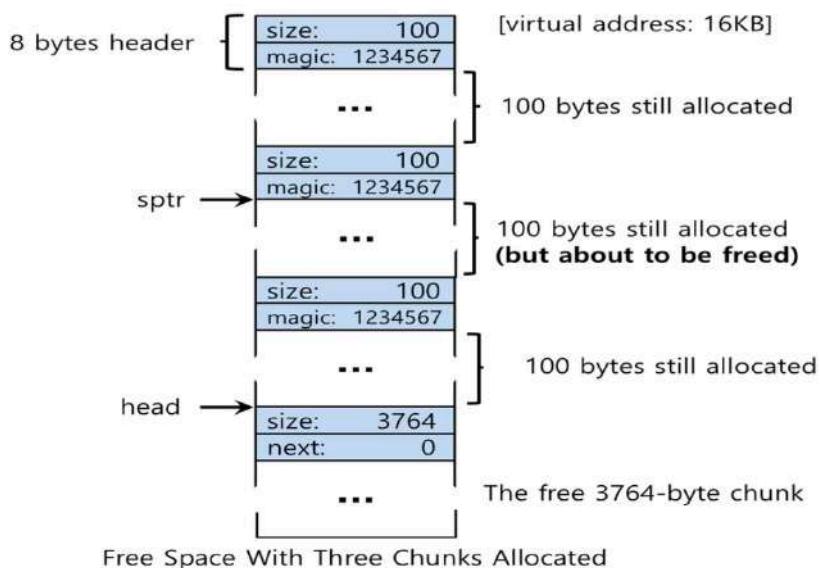
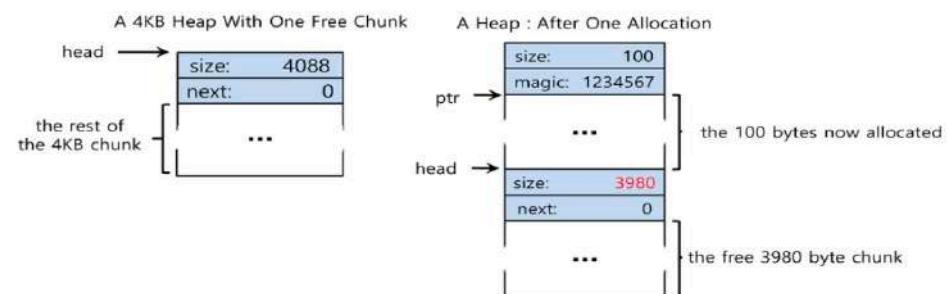
\* Now the status of the list is →



\* a request for 100 bytes by \*ptr = malloc(100)

- allocating 108 bytes out of the existing one free chunk.

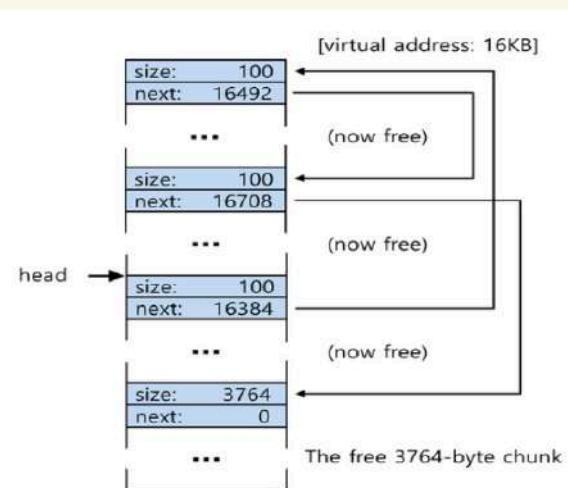
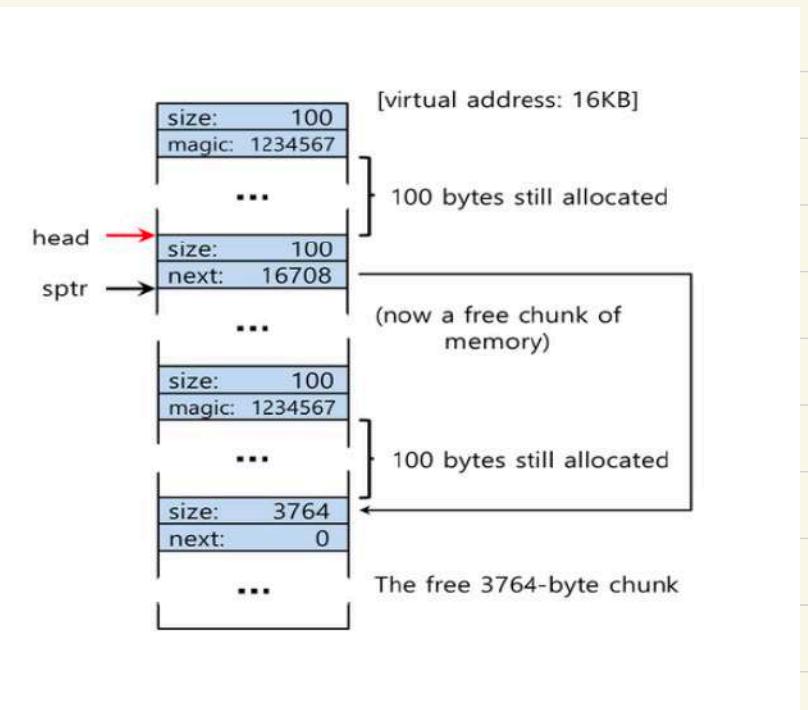
- shrinking the one free chunk to 3980 ( $4088 - 108$ )



\* Eg.- `free(sptr)`

- the 100 bytes chunks is back into the free list
- the free list will start with a small chunk and the list header will point the small chunk.

\* Assume that last two in-use chunks are also freed.

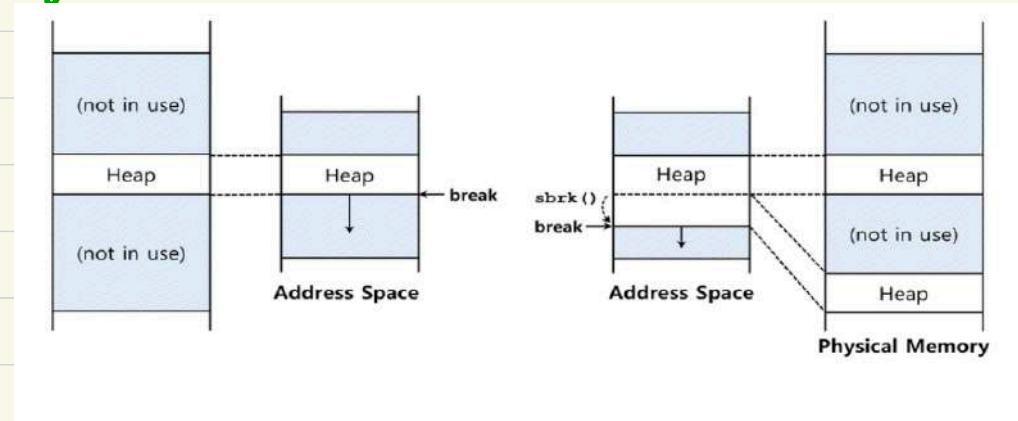


→ External fragmentation occurs  
- coalescing is needed in the list.

## 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.



## Basic strategies for managing free space

An ideal allocator → fast and minimizes fragmentation.

Best fit: finds free chunks that are big or bigger than the requested size.

return the one that is the smallest in that group of candidates.

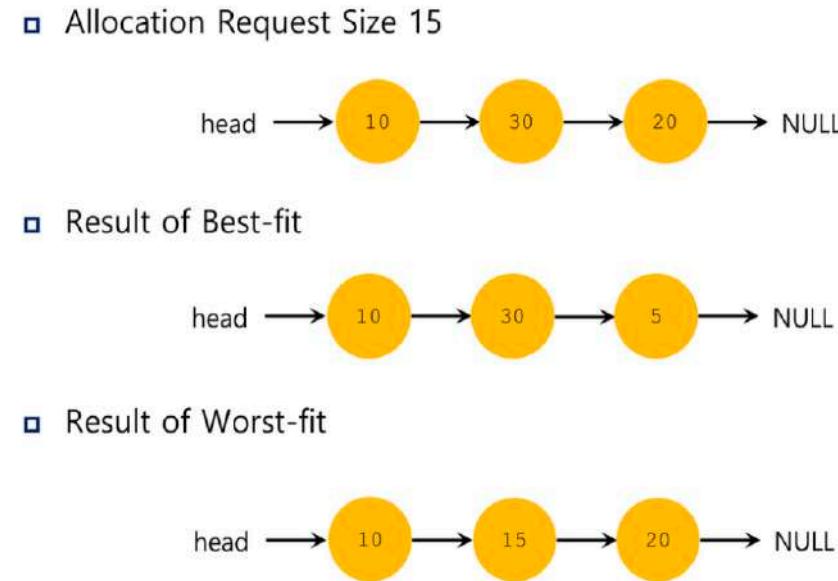
\* reduced waste of space

\* not fast, since it does an exhaustive search for the correct block.

worst fit: finds the largest chunk and return the requested amount; keep the remaining (large) chunk on the free list.

- \* leaves big chunks free instead of lots of small chunks that can arise from best-fit.
- \* full search of free space → costly!

eg:-



first fit: finds the first block that is big enough and returns the requested amount to the user.

- \* faster (as there is no exhaustive search)
- \* pollutes the beginning of the free list with small objects -

Next fit: Instead of always starting the search from the beginning of the list, next fit algorithm keeps an extra pointer to the location within the list where one was looking last.

\* spread the searches for free space throughout the list more uniformly.

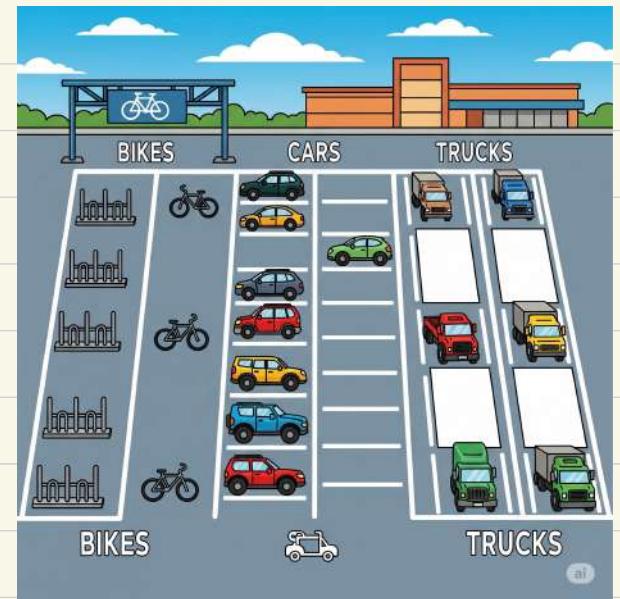
\* leaves small usable holes in the earlier parts of memory; over time leads to external fragmentation.

## Other approaches

### Segregated list

Some applications might have one (or a few) popular-sized request that it makes.

So, instead of keeping a single free list (which might contain free blocks of all sizes), the allocator keeps multiple free lists, each dedicated to a particular size class of memory blocks.



- This avoids having to search through lots of irrelevant blocks when looking for a free space.

simple segregation (Fixed-size classes)

- exact matches handled in size-specific regions  
(no internal fragmentation)
- everything else handled in a general-purpose pool.

Challenge? How much memory should one dedicate to the pool of memory that serves specialized requests of a given size, as opposed to general pool?

↳ slab allocator (originally designed for Solaris kernel)

- \* no static partitioning.
- \* instead, for each object type /size, the allocator maintains a cache made of slabs (chunks of contiguous memory).
- \* if demand grows, for that size, the allocator adds more slabs dynamically.
- \* if demand shrinks, it can release empty slabs back to the general pool.
- \* thus it automatically adapts to the workload, instead of requiring the allocator designer to guess sizes upfront -

Note that when you allocate a new object in a typical heap allocator, you need to do,

(constructor)

① Initialization: eg:- zeroing memory, setting default values, etc..  
For kernel objects, this can involve non-trivial work!

Doing this every time memory is allocated adds CPU overhead, especially when many allocations happen per second.

② Destruction : freeing objects may require cleaning up fields. This too is costly if done repeatedly.

Slab allocator's solution: Pre-initialized objects.

- Slabs are created in advance with all objects pre-initialized.  
i.e., when you allocate an object from the slab,
  - no need to redo the initialization
  - Object is immediately ready for use.

||| by when you free it

- Object can optionally be reset, but the structure of the object stays intact.
- To summarize, the constructor runs once per slab (when it's created)  
The destructor runs only if the slab is destroyed, not on every free.

## Buddy allocator

↳ an approach in which coalescing is simple.

Binary buddy allocator: free memory  $\rightarrow$  big space of  $2^N$ .

when a request comes in, the search for free space, recursively divides free space by two, until a block that is just big enough to accommodate the request is found.

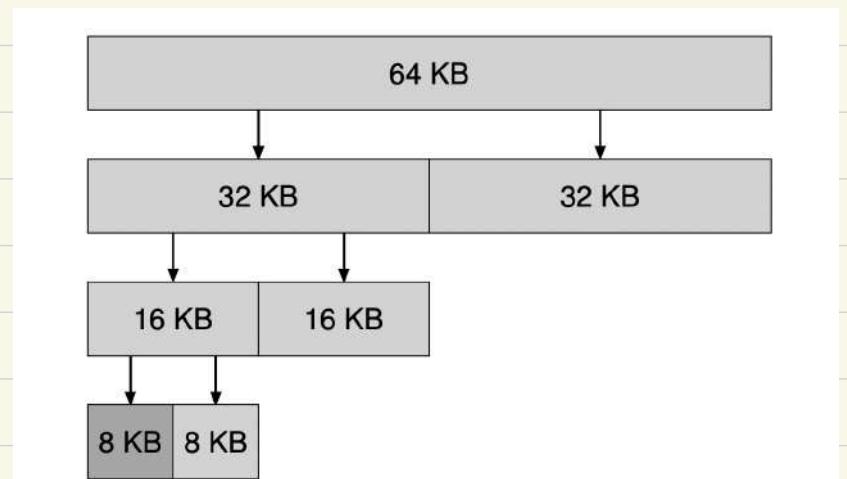
e.g.: free space  $\rightarrow$  64 KB, request  $\rightarrow$  7 KB block.

left most 8 KB block gets allocated.

Internal fragmentation can happen!  
(think, why?)

\* when the allocated 8 KB gets freed,  
check whether the "buddy" 8 KB is free;  
if so, coalesce those two blocks.

continue this recursive coalescing up the tree as far as possible.  
(either you restore the entire free space or find a buddy in use).



How to determine the buddy of a particular block?

If a block of size  $2^k$  starts at address A!

Buddy address =  $A \text{ XOR } 2^k$

- XOR flips the  $k^{\text{th}}$  bit of the address

why it works? - splitting always divides memory into halves aligned on  $2^k$ .

- flipping the  $k^{\text{th}}$  bit moves you from one half to another.

\* Buddy system relies on the property to merge blocks efficiently.  
(as no search needed  $\rightarrow$  just XOR)

Problem with all of the above approaches  $\rightarrow$  lack of scaling  
(searching the free list for a free block is slow).

↳ need to use advanced data structures  
to overcome this bottleneck.)

like, balanced binary trees, ↴ splay trees etc...