chapter 17
Free-Space Management

Kumo B4 massu (Yuichi Masuda)

What do we study in this chapter?

- The method to manage free-space of memory that are requested with variable-sized.
- Basic mechanisms to build general allocators.
- Strategies to build and use allocators efficiently.

Word List

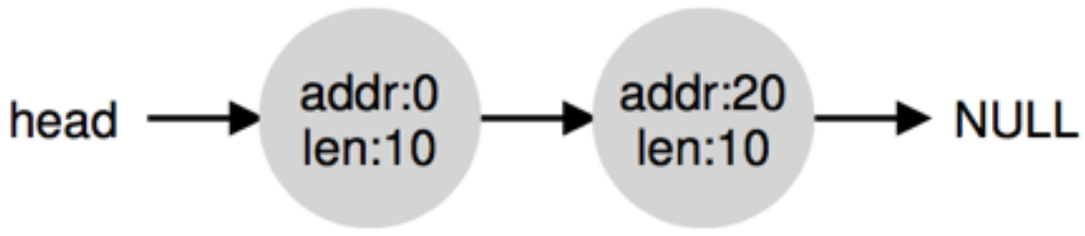
- allocator
- fragmentation
- chunk
- heap
- malloc()
- free()

Mechanisms of Allocator

Allocator is using mechanisms that are listed below.

- Splitting
- Coalescing
- Tracking the size of allocated regions
- Embedding a free list

Splitting



If the user wants to 1 byte free space, the allocator will find a free chunk that can satisfy the request and split it into two.

First chunk is returned to the user and Second chunk will remain on the list.

Coalescing

In this status, we can't use over 10 sizes.

Thus we use coalescing.

Tracking The Size Of Allocated Regions①

This issues is solved that allocator store a little bit of extra information in a header block which is kept in memory.

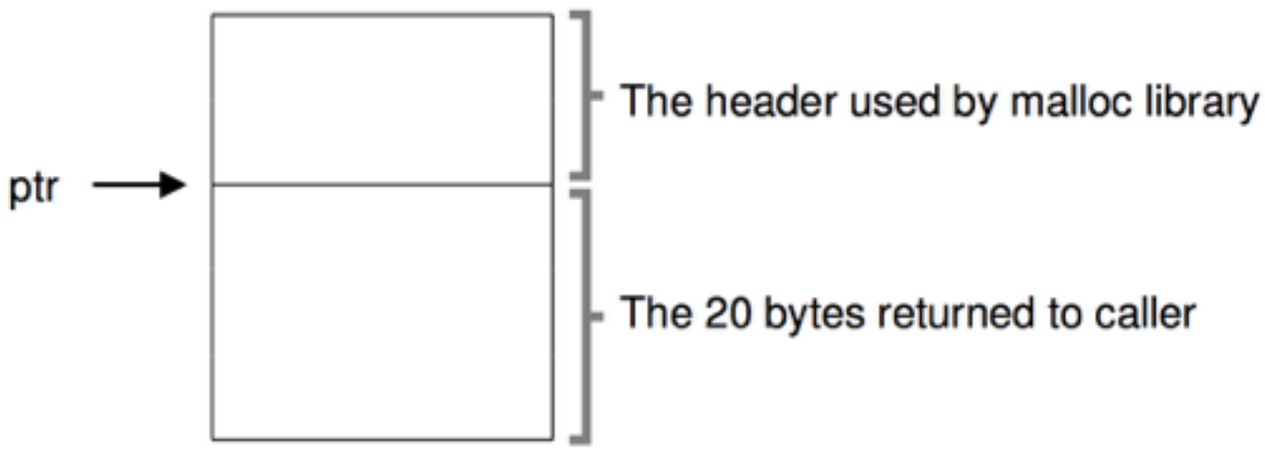


Figure 17.1: **An Allocated Region Plus Header**

ptr = malloc(20);

Tracking The Size Of Allocated Regions②

The header can have some information.
For example, additional pointer, magic number and other information.

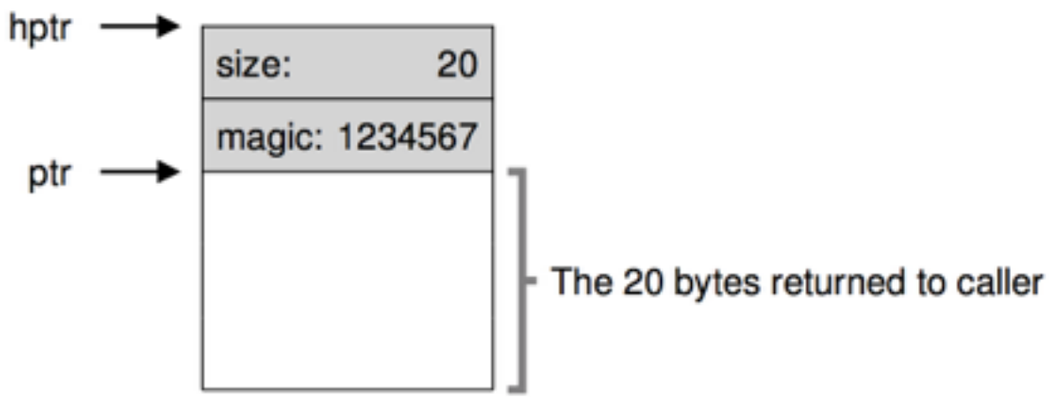


Figure 17.2: **Specific Contents Of The Header**

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

Tracking The Size Of Allocated Regions③

If the user calls “free(ptr)”, the allocator library return a simple pointer to figure out where the header begins.

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

When the user want N bytes of memory,
the allocator don’t search N bytes free space. It need search free space size of “N + the size of header” .

Embedding A Free List①

The memory free list exist inside the free space itself.

Assume that we a 4096-byte memory to manage.

Assume that heap is built within some free space acquire via a call “mmap();”

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

```
// 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;
```

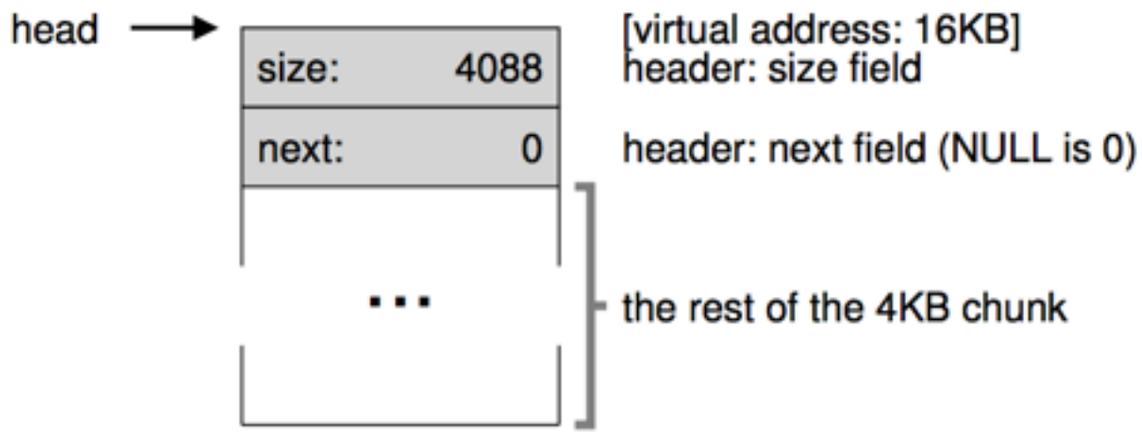


Figure 17.3: **A Heap With One Free Chunk**

Embedding A Free List②

The situation when the user request 100 bytes.(Assuming there is an 8 bytes header.)

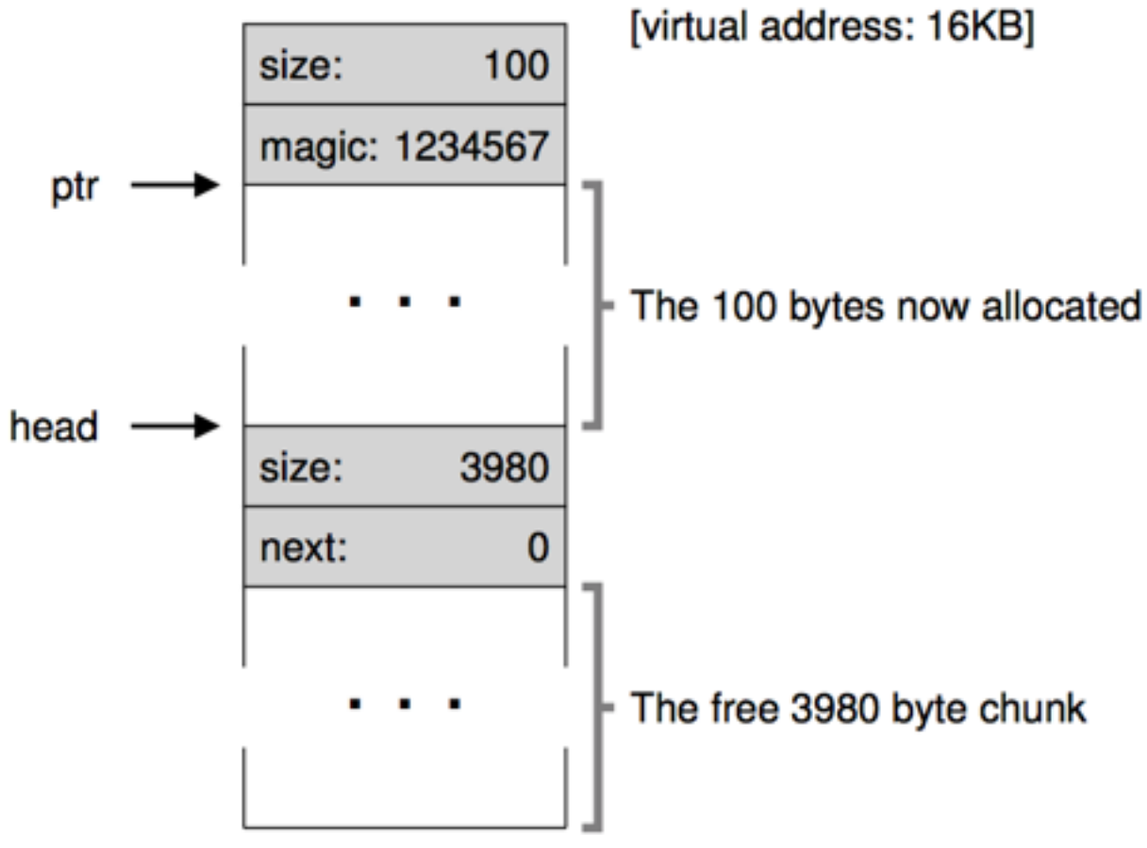


Figure 17.4: **A Heap: After One Allocation**

① The allocator find a chunk that is large enough to accommodate the request.

② The chunk will be split into two: one chance big enough to service the request, and the remaining free chunk.

Embedding A Free List③

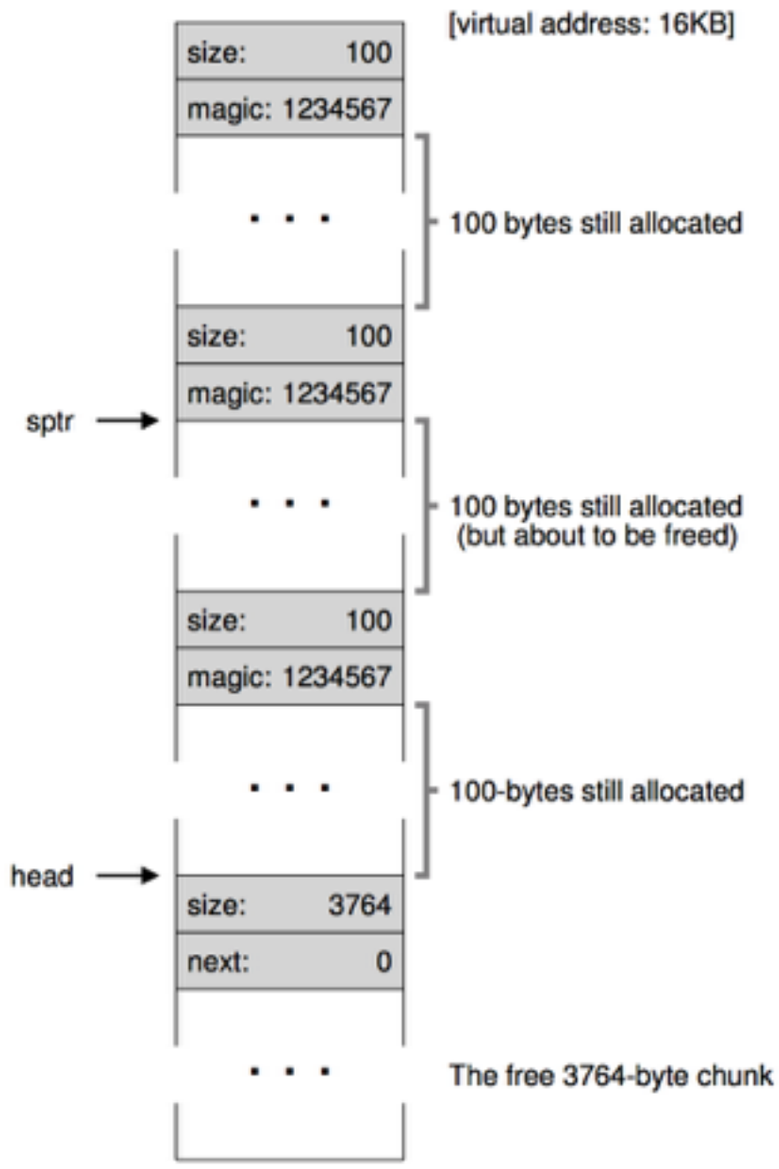


Figure 17.5: **Free Space With Three Chunks Allocated**

This figure shows a example when programs have three allocated regions, each of 100 bytes.

Embedding A Free List④

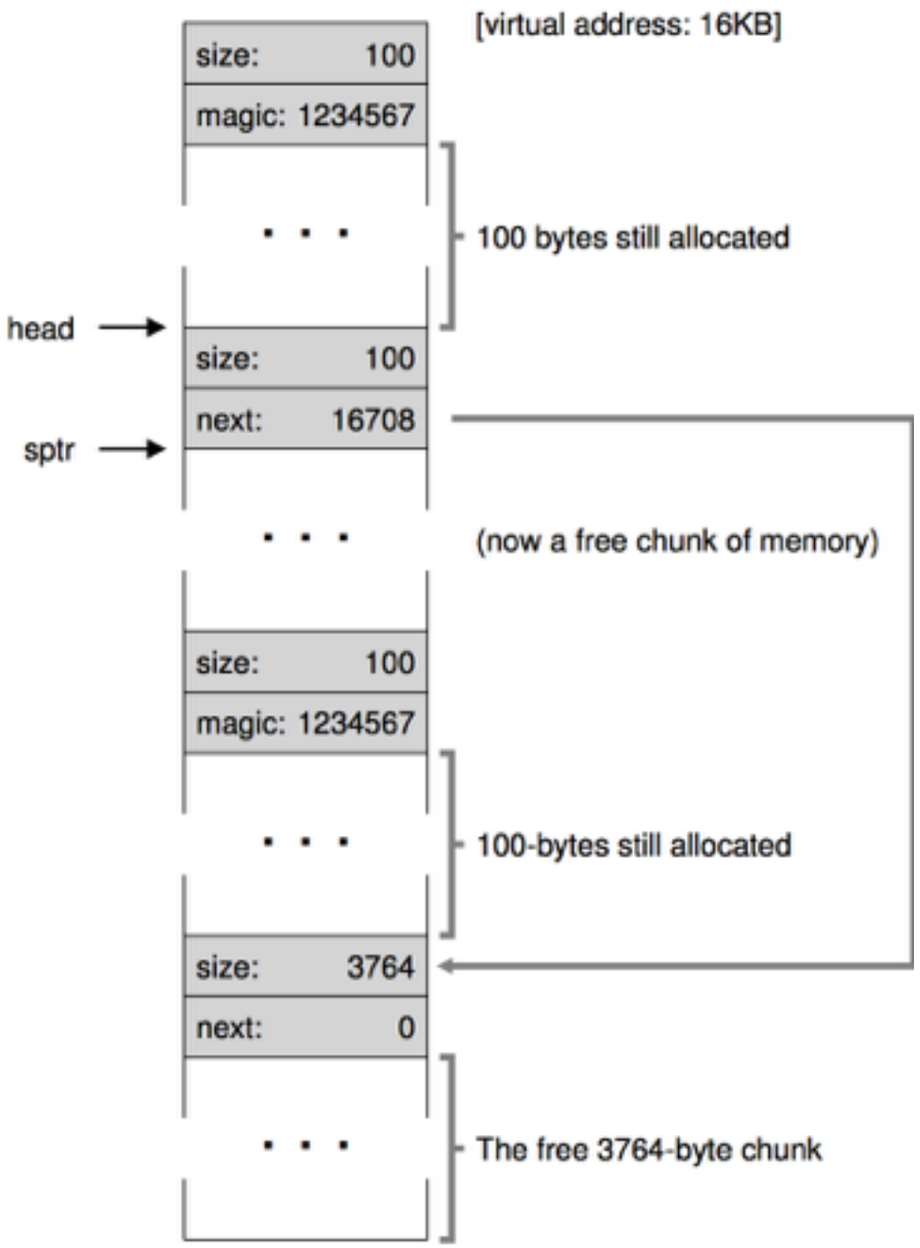


Figure 17.6: **Free Space With Two Chunks Allocated**

If the user calls “free(16500;)”, chunks become like a left figure.

Embedding A Free List⑤

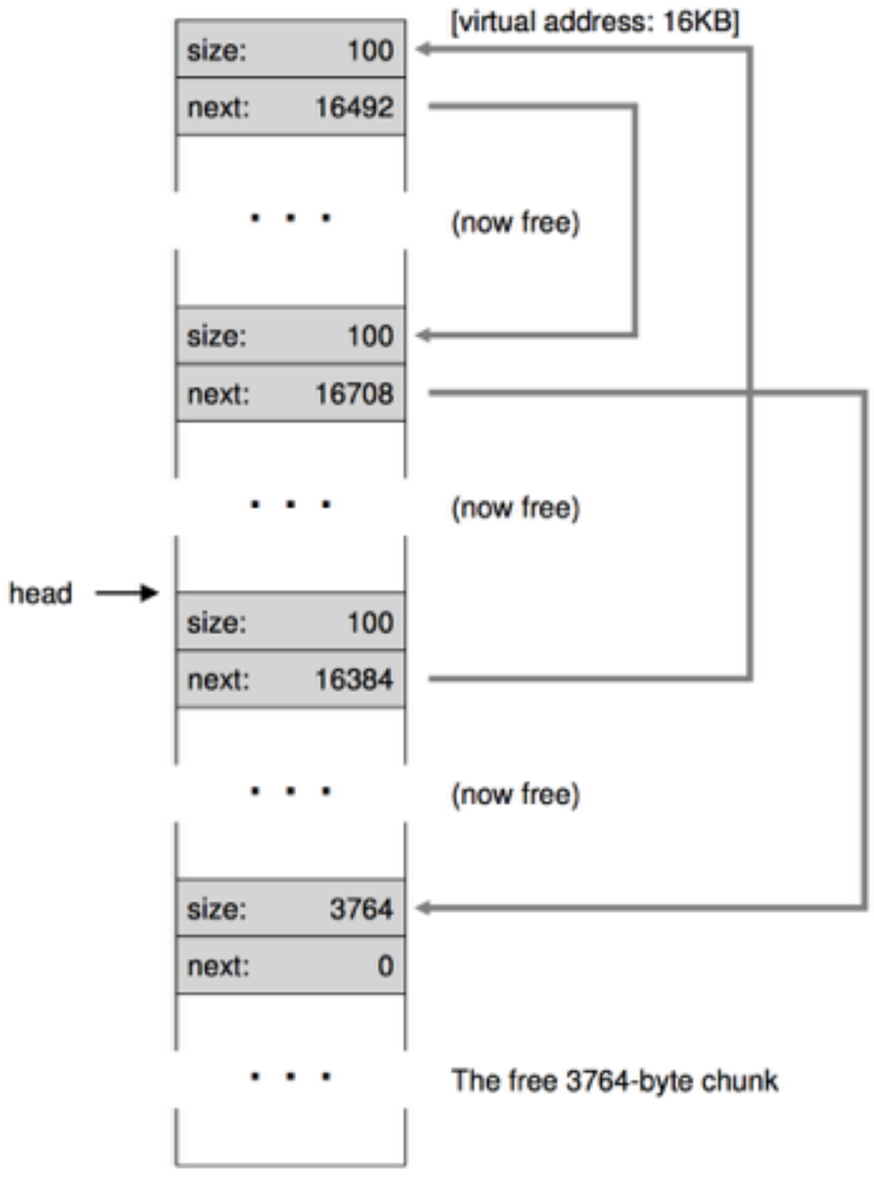


Figure 17.7: **A Non-Coalesced Free List**

If the allocator repeats the processes that I introduced so far, the free space is fragmented.

So it has to do “coalescing”.
Go through the list and merge neighboring chunks.

Strategies①

Best Fit

Find chunks of free memory that are as big or bigger than requested size and return the smallest one in that group.

Worst Fit

Find the largest chunks and return the requested amount.

These are the exhaustive search.

Strategies②

First Fit

Find the first block that is big enough and returns the requested amount to the user.

Next Fit

Next Fit is similar to First Fit. The difference is beginning position to find chunk.

These are not the exhaustive search.

Strategies③

Examples

Assume an allocation request size of 15.

Best Fit

Worst Fit

First Fit

Segregated Lists

- If a particular application has one poplar-sized request that it makes, keep a separate list just to manage objects of that size.
- If a such as memory is freed, it isn't initialized.
- This solution can reduce overheads that cause by initialization and destruction.

Buddy Allocation

The search for free space recursively divides free space by two until a block that is big enough to accommodate the request is found.