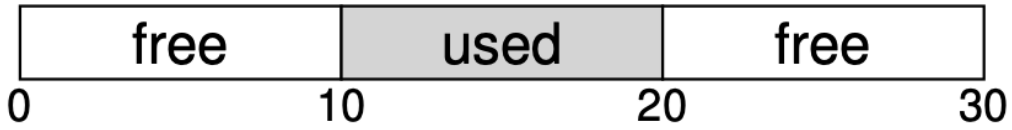
Chapter 17 Free-Space Management

Free-space management becomes difficult

- When free space consists of variable-sized units
- Arises
  - User-level memory-allocation libraries (malloc() and free())
  - OS managing physical memory when using segmentation to implement virtual memory
- Problem: external fragmentation
  - Not enough contiguous space for allocation request



Example: Total free space available 20 bytes

Fragmented into two chunks of size 10 each

Request: 15 bytes — fails even though there are 20 bytes free

Q: How to manage free space, when satisfying variable-sized requests?

Q: What strategies minimize fragmentation?

Q: What are the time and space overhead of the different approaches?

17.1 Assumptions

Basic interface:

- void * malloc(size)
- free(void *ptr)

The memory manager

- knows the size from the previous allocation
- Keeps a free list

(free list refer to a generic data structure to manage free space…)

- once a memory region is given to a process;
  - Can’t take back the memory that was given to the proc.
  - No compaction of free space is possible

Focus on external fragmentation

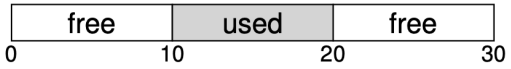
Can also have *internal fragmentation*; if a program requests more memory than what it needs then the “unused” memory goes to waste!

17.2 Low-level Mechanisms

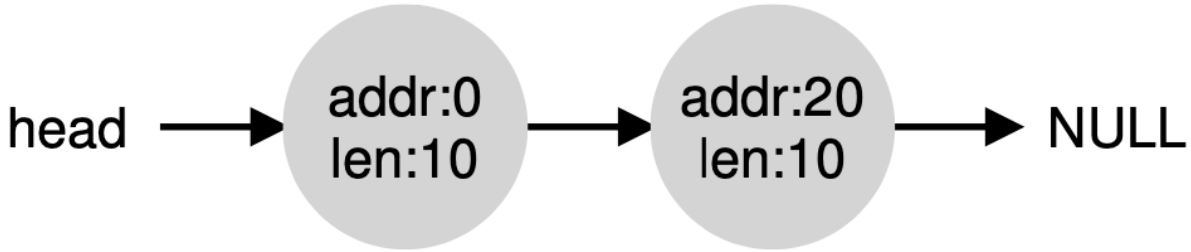
Splitting and Coalescing

Def. **Free list**: *set of elements describing free space still in the heap.*

Example: Free list for this heap:



- One entry describe the first 10-byte free segment (bytes 0-9)
- One entry for the other free segment (bytes 20-29)



Req A: 11 bytes: fails!

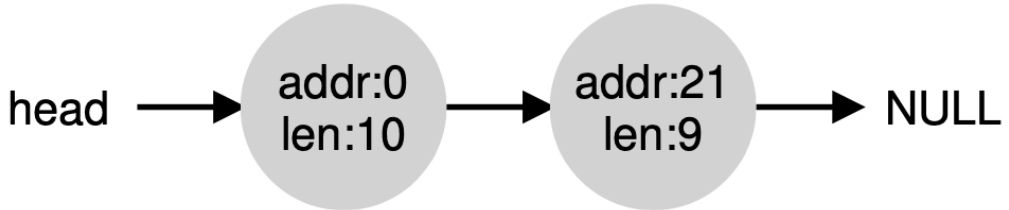
Req B: 10 bytes: both free chunks could be used!

Req C: 1 byte?

Memory allocator perform action know as **splitting**:

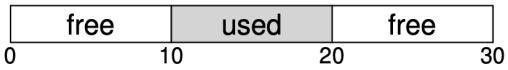
- Find a free chunk that satisfies the request and split it into two
  - Return the first chunk to the caller (1 byte)
  - Second chunk remain on the free list (9 bytes)

Free list after splitting could look like this:

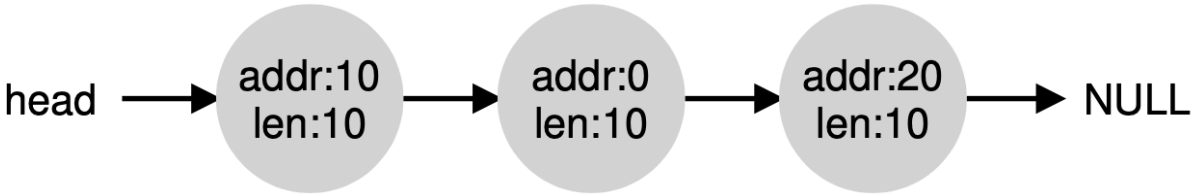


Coalescing of free space:

Example from the original heap allocation



Now if user calls free(10)



Problem:

- Entire heap is free
- But divided into three chunks of 10 bytes each
- Request for 20 bytes would still fail.

Solution: coalesce free space (when a chunk of memory is freed)

Look at the neighboring chunks and merge them into a single larger free chunk.

Tracking the Size of Allocated Regions

free(void *ptr):

Q: Given given a pointer, how can the malloc library quickly
determine the size of the chunk being freed.
(and incorporate the space back into the free list)

To accomplish this task:

Allocator store extra info in a **header** block.
Header block is also in memory, but just before the “handed-out” chunk of memory…

Example: Fig 17.1:

ptr = malloc(20);

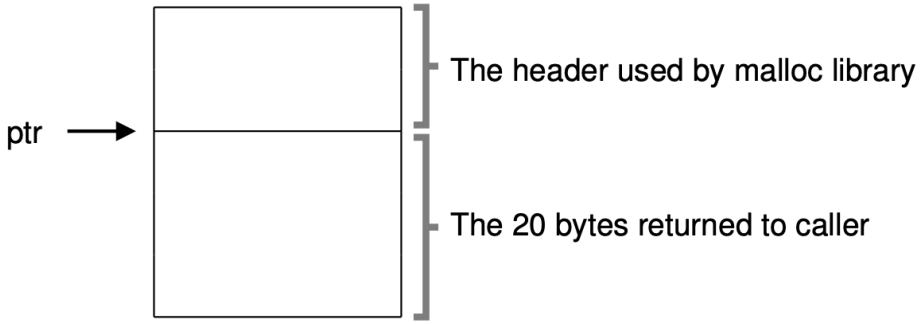


Figure 17.1: **An Allocated Region Plus Header**

Header contains:

- Size of allocated region (20 bytes in this case)
- Additional pointers to speed up deallocation (freeing)
- Magic number: integrity checking
- And other information

Example (simplified): Fig. 17.2

```
type header struct {
        size int
        magic int
}
```

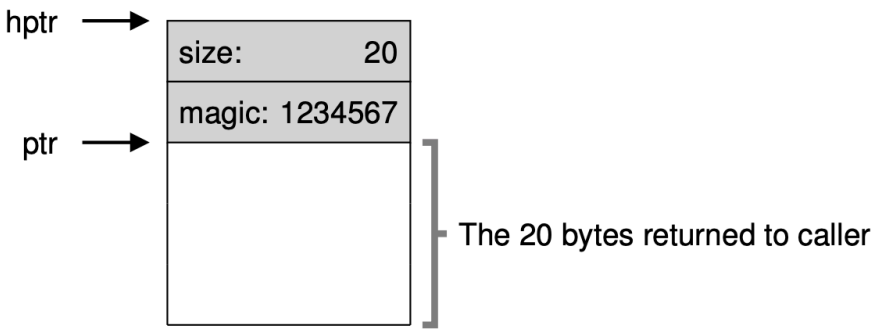


Figure 17.2: **Specific Contents Of The Header**

When process calls free(ptr):

Allocator uses simple pointer arithmetic to find the header:

```
void free(void *ptr) {
        header *hptr = (void *) ptr - sizeof(header);
        header.size to get the size of the allocated memory
}
```

Note:

- With a request for *N* bytes:
- Allocator actually allocates (*N*+sizeof(*header*)) bytes.
- The overhead is sizeof(*header*)

Embedding a Free List

Q: How do we keep a free list in the free space itself?

Example. Managing 4096-byte chunk of heap memory.

```
type node_t struct {
        size            int             // 4 bytes
        node_t          *next           // 4 bytes
}
```

Code to create a heap using the mmap() system call:

```
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;
```

After this code, we have a heap with one free chunk.

Size of heap: 4088 = 4096 - sizeof(node_t)

Heap ptr at virtual address: 16 KB

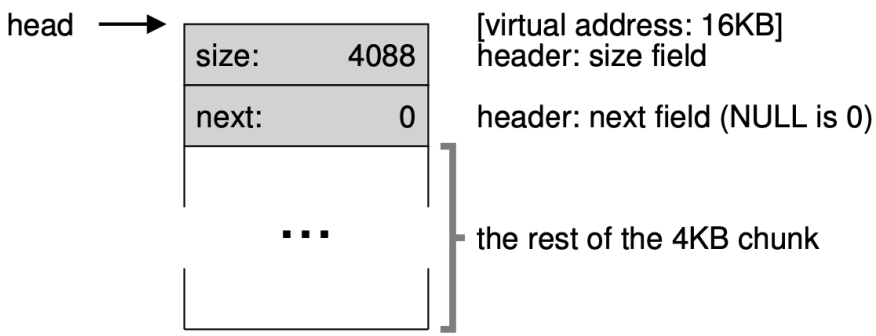


Figure 17.3: **A Heap With One Free Chunk**

Example: Request for 100 bytes

- Only one chunk of size 4088
- So this chunk is chosen
- Split chunk in two
  - One for the 100 bytes + sizeof(header) = 108
  - Remaining part is the free chunk

Assuming 8-byte header,
the heap now looks like Fig 17.4

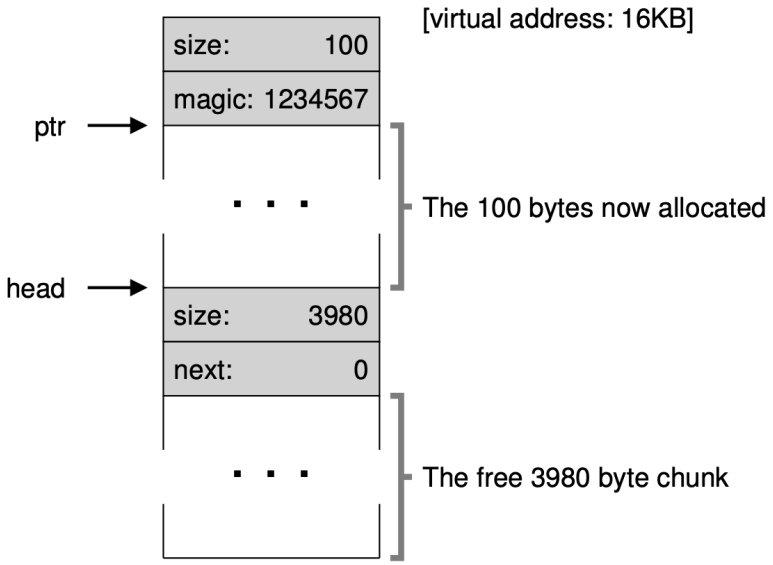


Figure 17.4: **A Heap: After One Allocation**

Example continues: Two more 100-byte allocations

Three allocations of 100 bytes or
108 bytes (incl. header)

Total 324 bytes of heap allocated memory
Head points to 16KB+324

Q: What happens when a process calls free?

Example. Calling free(16500);

- Start of memory region 16384 (16KB)
- 16500 = sptr in Fig 17.5 / 17.6
  - This is the pointer returned by the second call to malloc()
  - 16500 - 16384 = 116 = 108 +8
    - 108 for the first malloc chunk
    - 8 bytes header for “this” chunk

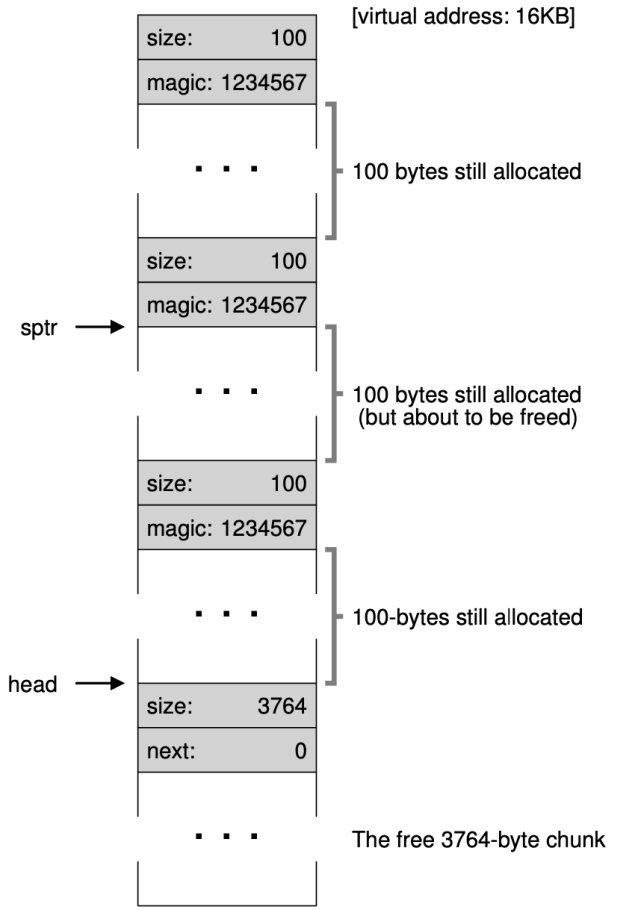


Figure 17.5: **Free Space With Three Chunks Allocated**

Allocator immediately finds the size of region to free.
(and adds the free chunk back on the free list).

Now head of list points to the first entry
(the 100 newly freed bytes)
The head node's next pointer points to
the next free chunk (16708)

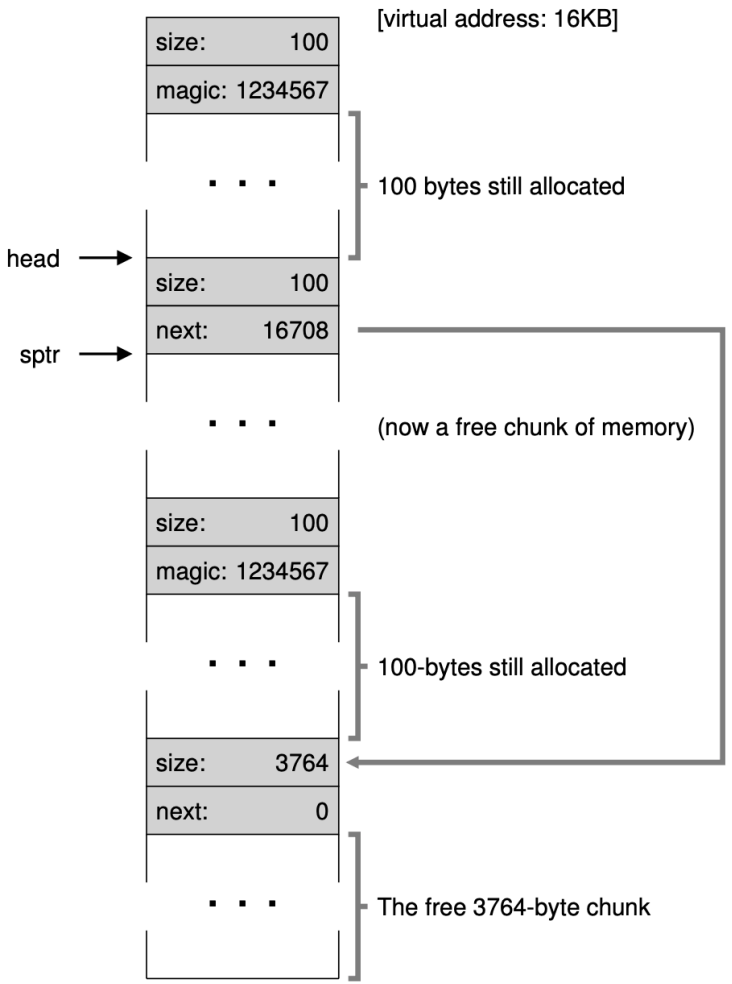


Figure 17.6: **Free Space With Two Chunks Allocated**

Fig. 17.7 shows all chunks are free —
free list is highly fragmented

Go through the list and merge neighboring chunks!

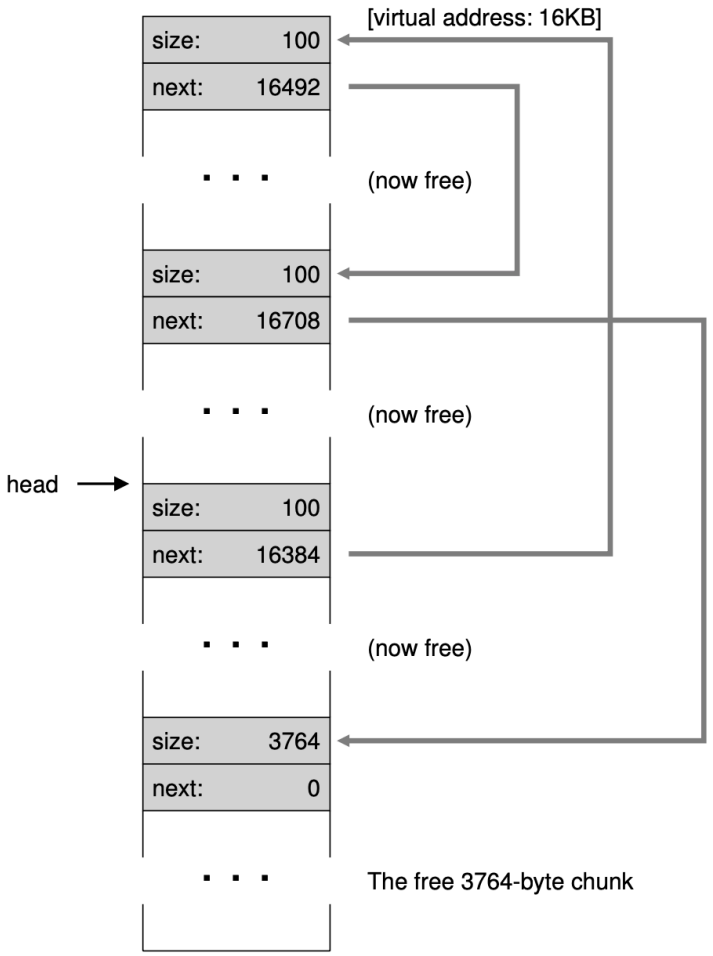


Figure 17.7: **A Non-Coalesced Free List**

To get back to the original heap with one free chunk; Fig. 17.3.

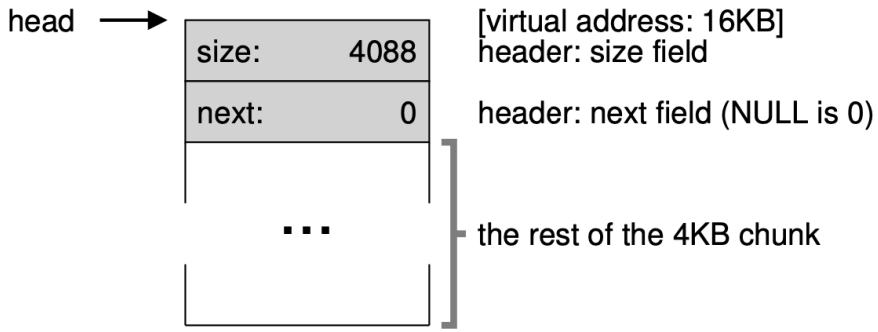


Figure 17.3: **A Heap With One Free Chunk**

17.3 Basic Strategies

Goal: *allocator should be fast and minimize fragmentation*

Challenge: allocations and free requests are arbitrary and not predictable

Best Fit

1. Search free list for free chunks >= requested size
2. Return the smallest chunk in the candidate set

Objective: reduce wasted space
Cost: naive implementations pay heavy performance penalty (require exhaustive search: O(N))
Drawback: can result in many small chunks (fragmentation)

Worst Fit

1. Find the largest chunk
2. Return requested amount

Objective: leave big chunks free instead of lots of small chunks
Cost: same as for Best fit.
Drawback: studies show it leads to excess fragmentation

First Fit

1. Find first chunk that is big enough
2. Returns requested amount

Objective: Fast
Cost: not exhaustive search
Drawback: pollutes beginning of free list with small objects

Next Fit

Keep extra pointer to the location in the list where one was looking last.
Start searching from this location.

Objective: spread searches uniformly over the list
Cost: same as for First Fit
Drawback: need an extra pointer

Examples

Here are a few examples of the above strategies. Envision a free list with three elements on it, of sizes 10, 30, and 20 (we'll ignore headers and other details here, instead just focusing on how strategies operate):

Assume an allocation request of size 15. A best-fit approach would search the entire list and find that 20 was the best fit, as it is the smallest free space that can accommodate the request. The resulting free list:

As happens in this example, and often happens with a best-fit approach, a small free chunk is now left over. A worst-fit approach is similar but instead finds the largest chunk, in this example 30. The resulting list: