【OS】Day20(2)

【Ch17】Free-Space Management(2)

17.2 Low-Level Mechanism(2)

Embedding a Free List

How do we build our free list inside the free space itself?

In a more typical list, when allocating a new node, we would just call `malloc()` when you need space for the node. Unfortunately, within the memory-allocation library, we cannot do this!

Instead, we need to build the list inside the free space itself.

Assume we have a 4096-byte chunk of memory to manage(i.e. the heap is 4KB). To manage this as a free list, we first have to initialize said list; initially, the list should have one entry, of size 4096(minus the header size). Here is the descrption of a node of the lsit:

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

Now let's look at some code that initializes the heap and puts the first element of the free list inside that space. We are assuming that the heap is built within some free space acquired via a call to the system call `mmap()`.

Here is the code:

```
//mmpa() 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;
```

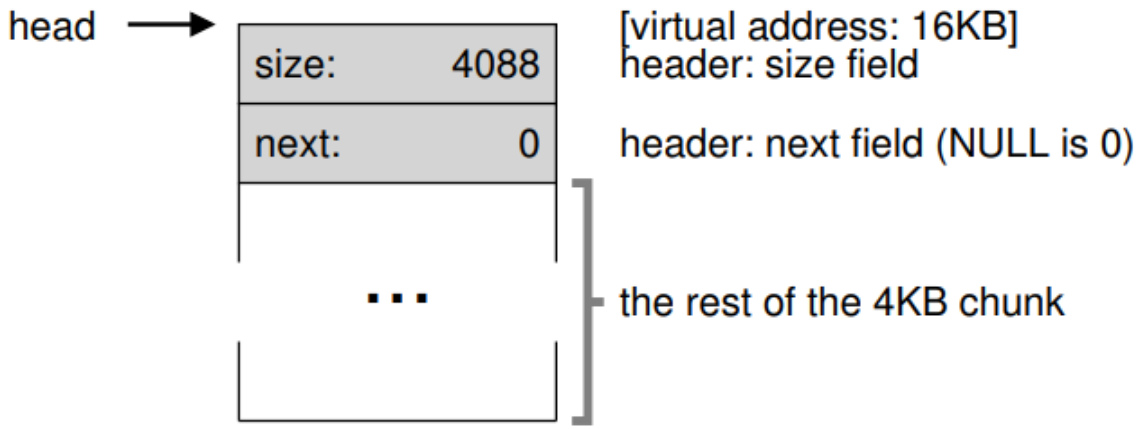

Figure 17.3: **A Heap With One Free Chunk**

After running this code, the status of the list is that it has a single entry, of size 4088. The header pointer contains the beginning addresss of this range.

Now, let's imagine that a chunk of memory is requested, say of size 100 bytes. To service this request, the library will first find a chunk that is large enough to accommodate the request; because there is only one free chunk(size: 4088), this chunk will be chosen.

Then, the chunk will be split into two: one chunk big enough to service the request(and header) and the remaining free chunk.

Assuming an 8-byte header(an integer size and an integer magic number), the space in the heap now looks like the figure below.

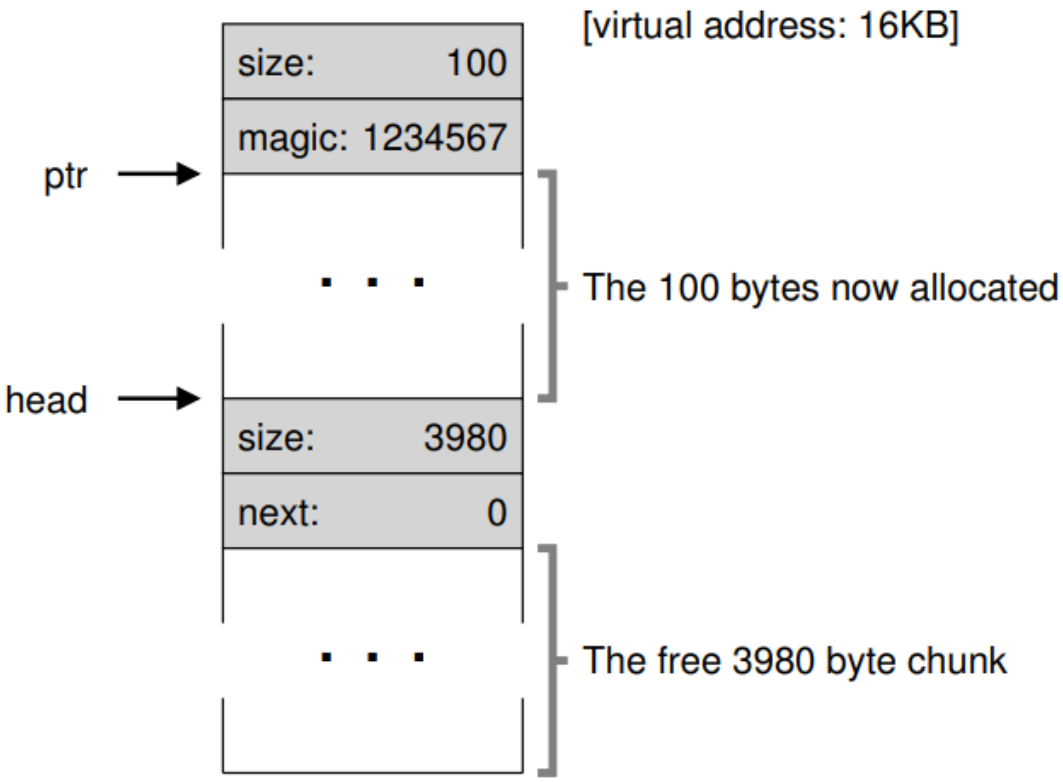

Figure 17.4: **A Heap: After One Allocation**

Thus, upon the request for 100 bytes, the library allocated 108 bytes out of the existing one free chunk, returns a pointer(marked ptr in the figure above) to it, stashes the header information immediately before the allocated space for later use upon `free()`, and shrinks the one free node in the list to 3980 bytes(4088 minus 108).

Now let's look at the heap when there are three allocated regions, each of 100 bytes(or 108 including the header). A visualization of this heap is shown below.

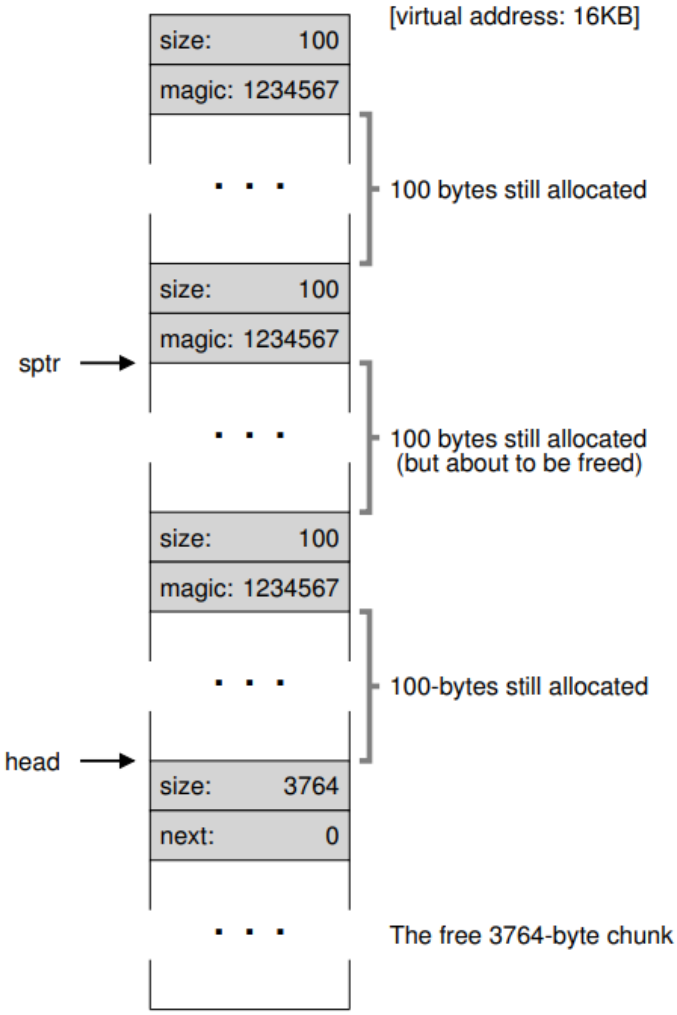

Figure 17.5: **Free Space With Three Chunks Allocated**

The first 324 bytes(3 x 108-byte chunks) of the heap are now allocated, and thus we see three headers in that space as well as three 100-byte regions being used by the calling program.

In this example, the application returns the middle chunk of allocated memory, by calling `free(16500)` (the value 16500 is arrived upon by adding the start of the memory region, 16384, to the 108 bytes of the previous chunk and the 8 bytes of the header for this chunk). This value is shown in the previous diagram by the pointer `sptr`.

The library immediately figures out the size of the free region, and then adds the free chunk back onto the free list. Assuming we insert at the head of the free list, the space now looks like this:

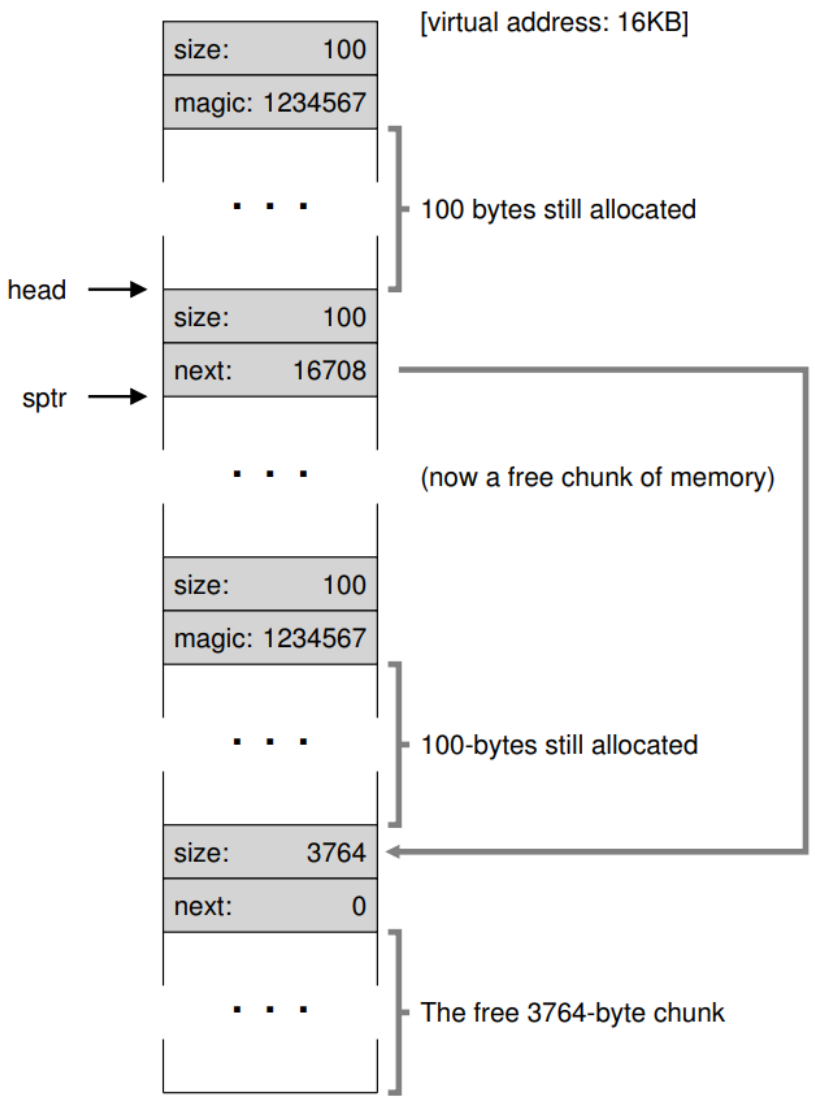

Figure 17.6: **Free Space With Two Chunks Allocated**

Now we have a list that starts with a small free chunk(100 bytes, pointed to by the head of the list) and a large free chunk(3764 bytes).

One last example: let's assume now that the last two in-use chunks are freed. Without coalescing, we end up with fragmentation.

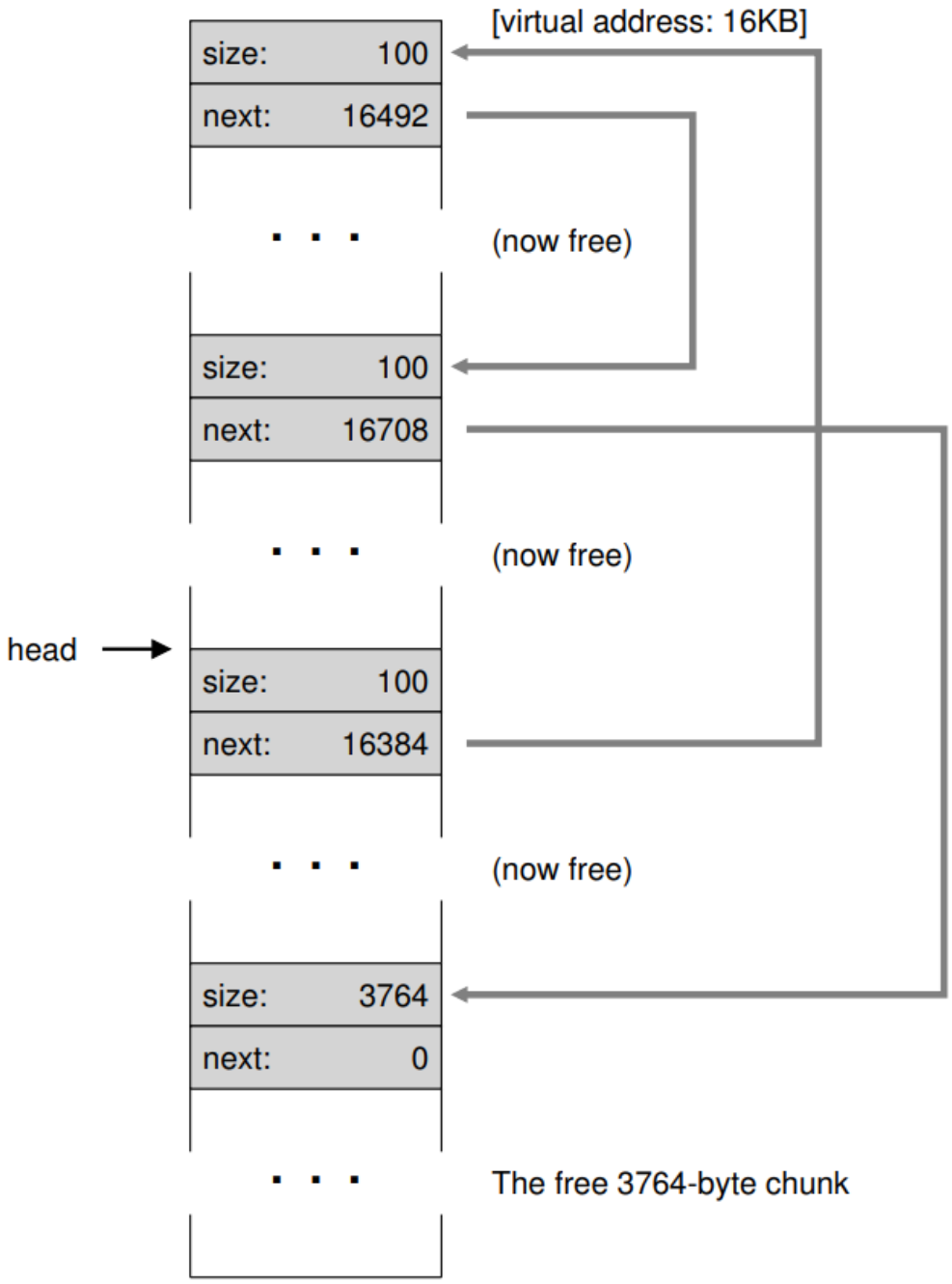

Figure 17.7: **A Non-Coalesced Free List**

As you can see from the figure, we now have a big mess. The list has not been coalesced yet.

Now we need to go through the list and merge neighbouring chunks; when finished, the heap will be whole again.

Growing the Heap

What should we do if the heap runs out of space?

The simplest approach is just to fail. In some cases this is the only option, and thus returning NULL is an honourable approach.