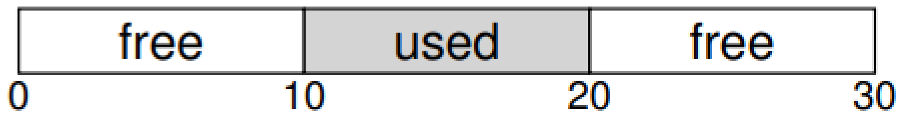
【OS】Day20

【Ch17】Free-Sapce Management

Mangaging free space can certainly be easy, as we will see when we discuss the concept of paging. It is easy when the space you are manging is divided into fixed-sized units; in such a case, you jsut keep a list of these fixed-sized units; when a client requests one of them, return the first entry.

Where free-space management becomes more difficult is when the free space you are managing consists of variable-sized units; this arises in a user-level memory-allocation library(as in `malloc()` and `free()`) and in an OS managing physical memory when using segmentation to implement virtual memory.

In either case, the problem that exists is known as external fragmentation: the free space gets chopped into little pieces of different sizes and is thus fragmented; subsequent requests may fail because there is no single contiguous space that can satisfy the request, even though the total amount of free space exceeds the size of the request.



In the example above, the total free space is 20 bytes; unfortunately, it is fragemented into two chunks of size 10 each. As a result, a request for 15 bytes will fail even though there are 20 bytes free.

17.1 Assumptions

We assume a basic interface such as that provided by malloc() and free().

- Specifically, `void *malloc(size_t size)` takes a single parameter, size, which is the number of bytes requested by the application; it hands back a pointer(of no particular type, or a `void` pointer in C) to a region of that size(or greater).

- The complementary routine `void free(void *ptr)` takes a pointer and frees the corresponding chunk.

Note the implication of the interface: the user, when freeing the space, does not inform the library of its size; thus, the library must be able to figure out how big a chunk of memory is when handed jsut a pointer to it.

The space that this library manages is known historically as the heap, and the generic data structure used to manage free space in the heap is some kind of free list.

This structure contains references to all of the free chunks of space in the managed region of memory.

Of course, this data structure need to be a list, but just some kind of data structure to track free space.

We further assume that primarily we are concerned with external fragmentation, as described above.

Allocators could of course also have the problem of internal fragmentation; if an allocator hands out chunks of memory bigger than that requested, any unasked for space in such a chunk is considered internal fragmentation waste and is another example of sapce waste.

We'll also assume that once memory is handed out to a client, it cannot be relocated to another location in memory.

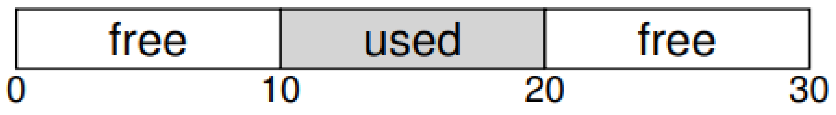
For example, if a program calls `malloc()` and is given a pointer to some space within the heap, that memory region is essentially "owned" by the program(and cannot be moved by the library) until the program returns it via a corresponding call to free(). Thus, no compaction of free space is possible, which would be useful to combat fragmentation.

Finally, we'll assume that the allocator manages a contiguous region of bytes.

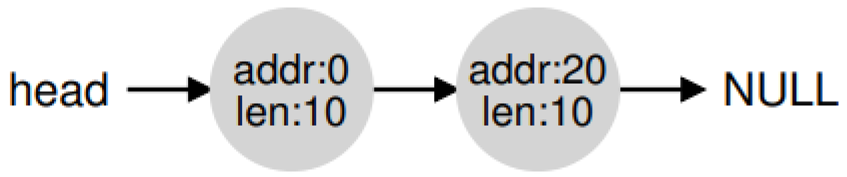
17.2 Low-level Mechanisms

Splitting and Coalescing

A free list contains a set of elements that describe the free space still remaining in the heap. Thus, assume the following 30-byte heap:



The free list for this heap would have two elements on it. One entry describes the first 10-byte free segment(bytes 0-9), and one entry describes the other free segment(bytes 20-29)

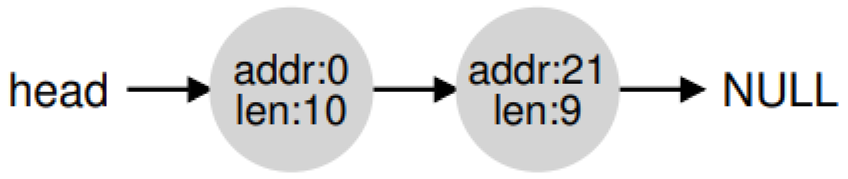


In this case, a request for exactly 10 bytes could be satisfied easily by either of the free chunks. *But what happens if the request is for something smaller than 10 bytes?*

Assume we have a request for just a single byte of memory. In this case, the allocator will perform an action known as splitting: it will find a free chunk of memory that can satisfy the request and split it into two.

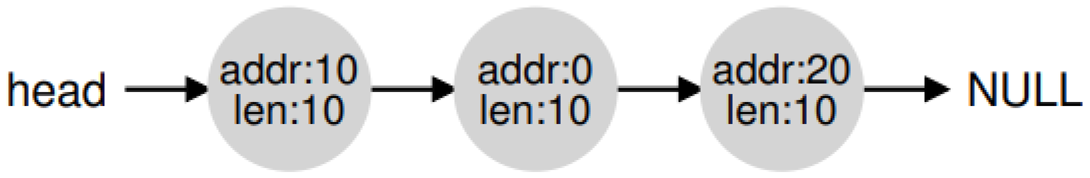
The first chunk it will return to the caller; the second chunk will remain on the list.

Thus, in our example above, if a request for 1 byte were made, and the allocator decided to use the second of the two elements on the list to satisfy the request, the call to `malloc()` would return 20 and the list would end up looking like this:



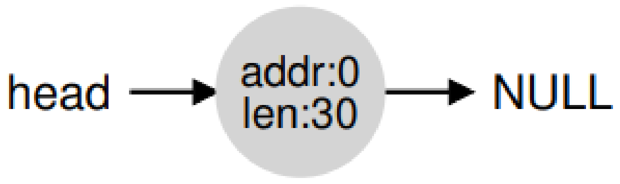
A corollary mechanism found in many allocators is known as coalescing of free space.

What happens when the caller above calls `free(10)` and thus returning the space in the middle of the heap? If we simply add this free space back into our list without too much thinking, we might en up with a list that looks like this:



Note the problem: while the entire heap is now free, it is seemingly divided into three chunks of 10 bytes each. Thus, if a user requests 20 bytes, a simple list traversal will not find such a free chunk, and return failure.

What allocators do in order to avoid this problem is coalesce free space when a chunk of memory is freed. The idea is simple: when returning a free chunk in memory, look carefully at the addresses of the chunk you are returning as well as the nearby chunks of free spae; if the newly-freed space sights right enxt to one(or two, as in this example) existing free chunks, merge them into a single large free chunk. Thus, with coalescing, our final list should look like this:



This is what the heap list looked list at first, before any allocations were made. With coalescing, an allocator can better ensure that large free extents are available for the application.

Tracking the Size of Allocated Regions

We noticed that the interface to `free(void *ptr)` does not take a size parameter; thus it is assumed that given a pointer, the malloc library can quickly determine the size of the region of memory being freed and thus incorportae the space back into the free list.

To accomplish this task, most allocators store a little bit of extra information in a header block which is kept in memory, usually just before the handed-out chunk of memory.

Let's look at an example. In this example, we are examining an allocated block of size 20 bytes, pointed to by ptr; imagine the user called `malloc()` and stored the results in `ptr`(e.g. `ptr = malloc(20)`)

The header minimally contains the size of the allocated region; it may also cotnain additional pointers to speed up deallocation, a magic number to provide additional integrity checking, and other information. Let's assume a simple header which contains the size of the region and a magic number, like this:

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

When the user calls `free(ptr)`, the library then uses simple pointer arithmetic to figure out where the header begins:

```
void free(void *ptr) {
  header_t *hptr = (header_t *)ptr - 1;
  ...
}
```

After obtaining such a pointer to the header, the library can easily determine whether the magic number matches the expected value as a sanity check(`asset(hptr→magic == 1234567)`) and calculate the total size of the newly-freed region via simple math.(i.e. adding the size of the header to size of the region).

Note the small but critical detail in the last sentence: the size of the free region is the size of the header plus the size of the space allocated to the user.

Thus, when a user requests N bytes of memory, the library does not search for a free chunk of size N; rather, it searches for a free chunk of size N plus the size of the header.