Chapter 17: Free-Space Management

In this chapter, we will specifically discuss the issues of free-space management. We will discuss the concept of **paging**, which is dividing the memory space into fixed-sized units.

**17.1 Assumptions:**

We assume a basic interface such as malloc() and free(). When free(), the library must figure out how big of a chunk of memory to free.

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**.

We assume that we primarily deal with **external fragmentation**. Internal fragmentation can also occur as the allocator hands out chunks of memory bigger than that requested. Any unasked space is considered internal fragmentation.

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

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

**17.2 Low-level mechanisms**

**Splitting and Coalescing:**

Consider the following 30-byte heap:

Shape

Description automatically generated

The free list of the heap would be:

Diagram

Description automatically generated

Any request that is greater than 10 bytes will now fail. Assume that we are requesting a byte of memory, then the allocator will perform a **splitting** action, which split a free chunk of memory that satisfies the request and split it in two. If it chose the second free chunk, the free list would now be:

Diagram

Description automatically generated

A corollary mechanism found in many allocators is **coalescing** of free space. Given the previous example, if the user calls free(10), the free list would now be:

A picture containing text, clock, device

Description automatically generated

The whole memory is free, but it is divided into three chunks, so if a user requests 20 bytes of memory, it would cause error. In such cases, the allocator will coalesce free space when a chunk of memory is freed, so the memory would look like this:

Diagram

Description automatically generated

**Tracking The Size Of Allocated Regions**

When we call free, the allocators must determine the size of the region in memory. To accomplish this task, most allocators store a little bit of extra information ins a **header** block.

Diagram

Description automatically generated

The above parts of the 20 bytes are the header block. It contains information about the size of the malloc and a magic number. When we call free(ptr), the library simply uses simple pointer arithmetic to figure out the header block to check if the magic number is the expected value and calculate the total size of the newly-freed region.

**Embedding a Free List**

To manage a chunk of memory as a free list, we first initialize said list. Each node of the list is described as:

Box and whisker chart

Description automatically generated with low confidence

Then, as we initialize the heap

Text

Description automatically generated

After we initialize the heap, the list has a single entry of size 4088. Then, as we allocate memory in the heap, header blocks will always be allocated

Diagram

Description automatically generated with medium confidence

But what happen when we free a chunk of memory? For example, we call free(16500) to free the second chunk (by adding 108 to 16384 and 8 bytes of header block). The library immediately figures out the size of the free region, and then adds the free chunk back onto the free list. The heap is fragmented

When we free every allocated memory, the heap would look like figure 17.6. Although every memory is freed, but it is divided into pieces. The solution is to **merge** neighboring chunks.

**Growing The Heap**

What should we do if the heap is running out of space? We will return NULL indicating that we failed. However, most traditional allocators start with a small heap, and then request more memory from the system when run out.

**17.3 Basic strategies**

The ideal allocator is both fast and minimizes fragmentation.

**Best fit**

First, search through the free list and find chunks of free memory that are as big or bigger than the requested size. Then, return the one that is the smallest in that group of candidates.

Graphical user interface, diagram

Description automatically generated

The reason is that best fit try to reduce waste. However, it must perform exhaustive search with naïve implementation.

**Worst fit**

It does the opposite to best fit. However, a full search of free space is required, and thus this approach can be costly. Worse, most studies show that it performs badly, leading to excess fragmentation while still having high overheads.

**First Fit**

Finds the first block that is big enough and returns the requested amount to the user. It has the advantage of speed as it requires no searching, but sometimes pollutes the beginning of the free list with small objects.

**Next Fit**

Instead of always beginning the first-fit search at the beginning of the list, the next fit algorithm keeps an extra pointer to the location within the list where one was looking last. Performance is similar to first fit because no searching required.

**17.4 Other Approaches**

If a particular application has one (or a few) popular-sized request that it makes, keep a separate list just to manage objects of that size; all other requests are forwarded to a more general memory allocator.

The benefit is that by having a chunk of memory dedicated for one particular size of requests, fragmentation is much less of a concern. In addition, requests can be served quickly without any search.

How much memory should be dedicated to specialized request? This problem is dealt with by the **slab allocator**

**Buddy Allocation**

This approach makes it is easier to coalesce.

In this a system, free memory is first conceptually thought of as one big space of size 2N. When a request for memory is made, the search for free space recursively divides free space by two until a block that is big enough to accommodate the request is found and return to the user.

This approach might suffer from internal fragmentation as we are only giving power-of-two-sized blocks.

However, when a block is freed, the coalescing process happens recursively in a tree, so it takes less time.

A picture containing diagram

Description automatically generated

This system works well because each block only differs by a single bit.

**Other ideas**

The previous approaches are lack of scaling. Specifically, searching a list is slow. Thus, advanced allocators use more complex data structure to address these costs.

Allocators are created to work well on multiprocessors system.