Free-Space Summary - SistOp

Lautaro Bachmann

Contents

Free-Space Management

Definitions

free-space management.

It is easy when the space you are managing is divided into fixed-sized units; Where free-space management becomes more difficult is when the free space you are managing consists of variable-sized units;

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.

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 once memory is handed out to a client, it cannot be relocated to another location in memory.

Assumptions

- We assume a basic interface such as that provided by malloc() and free().
  - This implies that the user, when freeing the space, does not inform the library of its size;
- We assume that primarily we are concerned with external fragmentation,

Low-level Mechanisms

Splitting and Coalescing

splitting:

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.

the split is commonly used in allocators when requests are smaller than the size of any particular free chunk.

coalescing of free space.

when returning a free chunk in memory, if the newlyfreed space sits right next to existing free chunks, merge them into a single larger free chunk.

Tracking The Size Of Allocated Regions

header block

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.

The header minimally contains the size of the allocated region; it may also contain additional pointers to speed up deallocation,

Embedding A Free List

you need to build the list inside the free space itself.

the library will first find a chunk that is large enough to accommodate the request;

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

out of the existing one free chunk, returns a pointer to it, stashes the header information immediately before the allocated space for later use upon free(),

When the calling program returns some memory via free()

The library immediately figures out the size of the free region, and then adds the free chunk back onto the free list.

Growing The Heap

what should you do if the heap runs out of space?

- The simplest approach is just to fail.
- Most traditional allocators start with a small-sized heap and then request more memory from the OS when they run out.

Basic Strategies

Best Fit

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;

naive implementations pay a heavy performance penalty when performing an exhaustive search for the correct free block.

Worst Fit

The worst fit approach is the opposite of best fit; find the largest chunk and return the requested amount; keep the remaining (large) chunk on the free list.

a full search of free space is required, and thus this approach can be costly.

First Fit

The first fit method simply finds the first block that is big enough and returns the requested amount to the user. First fit has the advantage of speed but sometimes pollutes the beginning of the free list with small objects.

address-based ordering;
One approach is to use address-based ordering; by keeping the list ordered by the address of the free space, coalescing becomes easier, and fragmentation tends to be reduced.

Next Fit

the next fit algorithm keeps an extra pointer to the location within the list where one was looking last. The idea is to spread the searches for free space throughout the list more uniformly,

The performance of such an approach is quite similar to first fit, as an exhaustive search is once again avoided.

Other Approaches

Segregated Lists

The basic idea
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.

By having a chunk of memory dedicated for one particular size of requests, fragmentation is much less of a concern;

New complications
how much memory should one dedicate to the pool of memory that serves specialized requests?

the slab allocator
when the kernel boots up, it allocates a number of **object caches** for kernel objects that are likely to be requested frequently;

the object caches thus are each segregated free lists of a given size and serve memory allocation and free requests quickly.

When a given cache is running low on free space, it requests some **slabs** of memory from a more general memory allocator when the reference counts of the objects within a given slab all go to zero, the general allocator can reclaim them from the specialized allocator,

The slab allocator also goes beyond most segregated list approaches by keeping free objects on the lists in a pre-initialized state. the slab allocator thus avoids frequent initialization and destruction cycles per object and thus lowers overheads noticeably.

Buddy Allocation

binary buddy allocator
free memory is first conceptually thought of as one big space of size 2^N

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. At this point, the requested block is returned to the user.

The beauty of buddy allocation
is found in what happens when that block is freed. The allocator checks if the buddy of the block is still free; if so, it coalesces those two blocks. This recursive coalescing process continues up the tree, either restoring the entire free space or stopping when a buddy is found to be in use.

Other Ideas

One major problem
with many of the approaches described above is their lack of **scaling.**

more complex data structures
searching lists can be quite slow. advanced allocators use **more complex data structures** to address these costs, trading simplicity for performance.

Data structures examples

- balanced binary trees,
- splay trees,
- partially-ordered trees

multiple processors
a lot of effort has been spent making allocators work well on multiprocessor-based systems.