13. Free Space Management

It's easy to manage
when the space you are managing is divided into fixed-sized units
: you just keep a list of these fixed-sized units.
when a client requests one of them, return the first entry

BUT, when the free space you are managing consists of **variable-sized** units : **DIFFICULT = external fragmentation**

generic data structure used to manage free space in the heap = some kind of **free list.**

- contains references to all of the free chunks of space in the managed region of memory

If allocator hands out chunks of memory bigger than that requesed => internal fragmenation

Splitting

: allocator finds a free chunk of memory that can satisfy the request and split it into 2.

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

Coalescing of free space when a chunk of memory is freed

When returning a free chink in memory,
If the newly freed space sits right next to existing free chunks,
merge them into a single larger free chunk
=> can better ensure that large free extents are available for the application

Tracking the size of allocated Regions

free() doesn't take a size parameter : HOW?
-> most allocators store a little bit of extra inforamtion 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
*coalescing free spaces also free used-to-be header space since it's only need 1 header.

Growing the Heap

What should you do if the heap runs out of space?
: just to fail.

Most traditional allocators start with a small-sized heap and then request more memory from the OS when the run out.

Typically, this means they make some kind of system call (ex. sbrk) to grow the heap, and then allocate the new chunks from there.

To service the sbrk request, the OS finds free physical pages, maps them into the address space of the requesting process, and then returns the value of the end of the new heap; at that point, a larger heap is available, and the request can be successfully serviced.

Basic strategies for managing free space

Best fit : smallest fit

by returning a block that is close to what the user asks, best fit tries to reduce wasted space.
However : naive implementations pay a heavy performance penalty when performing an exhaustve search for the correct free block

Worst Fit : opposite of best fit.

find the largest chunk. 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 fragmetation while still having high overheads.

First fit

adventage of speed - no exhaustive search of all the free spaces
but sometime pollutes the beginning of the free list with small objects.
=> how the allocator manages the free list's order becomes an issue.
Ex. 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

When you do next allocation, start searching from that point, not from the beginning.
to spread the searches for free space throughout the list more uniformly, thus avoiding splintering of the beginning of the list.
The performance of such an approach is quite similar to first fit, as an exhausive search is once again avoided.

the difference is in the search cost
both best-fit & worst fit : look through the entire list
first fit only examines free chunks until it finds one that fits, thus reducing search cost.

Segregated Lists

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.

Benefits :

- by having a chunk of memory dedicated for one particular size of requests, fragmentation is much less of a concern
- allocation and free requests can be served quite quickly when they are of the right size, as no complicated search of a list is required.

Ex. slab allocator
when the kernel boots up, it allocates a number of object caches for kernel objects that are likely to be requested frequently (such a slocks, file-system inodes, etc).
the object caches thus are each segregated free lists of a given size
=> serve memory allocation & free requests quickly.

the slab allocator also goes beyond most segregated list approaches by **keeping free objects on the lists in a pre-initialized state**.

- initialization & destruction of data structures is costly.

by keeping freed objects in a particular list in their initialized state, the slab allocator thus avoids frequent initialization & destruction cycles per object => thus lowers overheads noticeably

Buddy allocation : making coalescing simple / 2^N
this scheme can suffer from internal fragmentation, as you are only allowed to give out power-of-2 sized blocks.

beauty : when block is freed,
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.

why it works so well : simple to determine the buddy of a particular block.
the address of each buddy pair only differs by a single bit.
which bit is determined by the level in the buddy tree.

lack of Scaling

: searching llsits can be quite slow
=> advanced allocators use more complex data structures,
trading simplicity for performance.
Ex. binary trees, splay trees, or partially-ordered trees