【OS】Day20(3)

【Ch17】Free-Space Management

17.3 Basic Strategies

Best Fit

The best fit strategy is quite simple: 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; this is the so called best-fit chunk.

The intuition behind best fit is simple: by returning a block that is close to what the user asks, best fit tries to reduce wasted space.

However, there is a cost; naïve 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 of the free list.

Worst fit tries to thus leave big chunks free instead of lots of small chunks that can arise from a best-fit approach.

Once again, 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

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-no exhaustive search of all the free spaces are necessary-but sometimes pollutes the beginning of the free list with small objects. One approach to solve this issue is to use the 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

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.

The idea is to spread the searches for free space throughout the list more uniformly, thus avoiding splintering of the beginning of the list.

17.4 Other Approaches

Segregated Lists

One interesting approach is the use of segregated lists. The idea is simple: 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; moreover, 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.

Buddy Allocation

Because coalescing is critical for an allocator, some approaches have been designed around making coalescing simple. One good example is found in the binary buddy allocator.

In such a system, 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.

Here is an example of a 64KB free space getting divided in the search for a 7KB block.

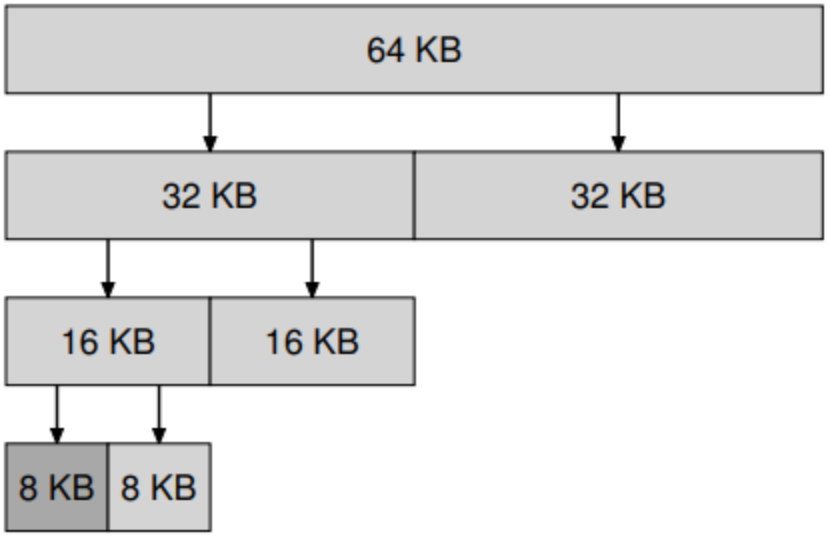

Figure 17.8: **Example Buddy-managed Heap**

Note: This system can suffer from internal fragmentation, as we are only allowed to give out power-of-two-sized blocks.

The beauty of buddy allocation is found in what happens when that block is freed. When returning the 8KB block to the free list, the allocator checks whether the “buddy” 8KB is free; if so, it coalesces the two blocks into a 16KB block. The allocator then checks if the buddy of the 16KB block is still free; if so, it coalesces those two blocks.