COMPILER DESIGN

→REDUCING FRAGMENTATION

→MANUAL DEALLOCATION REQUESTS

WHAT IS FRAGMENTATION ?

→Fragmentation is an unwanted problem where the memory blocks cannot be allocated to the processes due to their small size and the blocks remain unused.

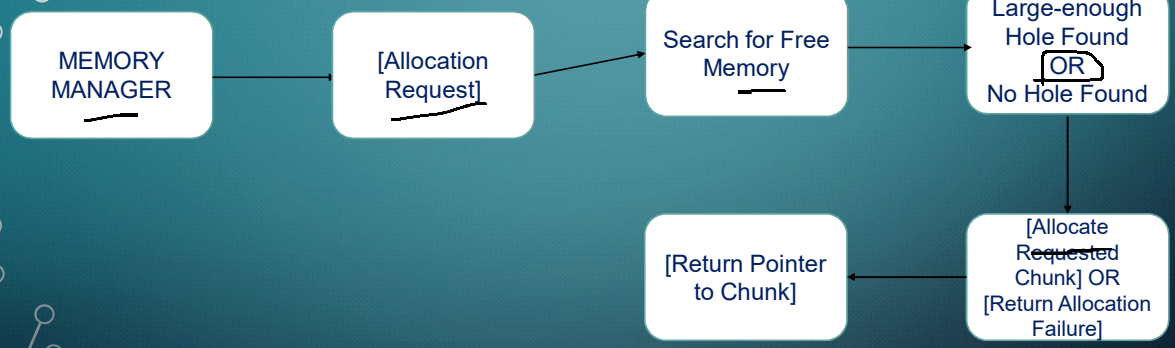
REDUCING FRAGMENTATION

→At the beginning of program execution, the heap is one contiguous unit of free space.

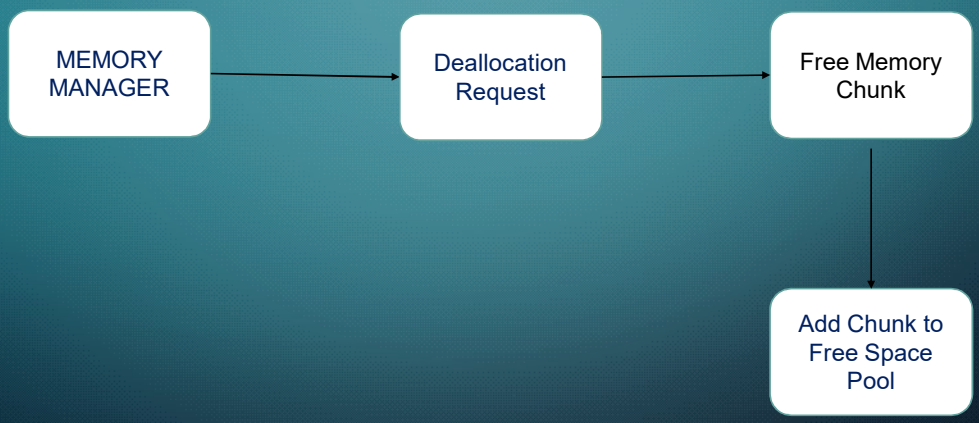
→As the program allocates and deallocates memory, this space is broken up into free and used chunks of memory.

→We refer to the free chunks of memory as holes.

→ WITH EACH ALLOCATION REQUEST, THE MEMORY MANAGER MUST PLACE THE REQUESTED CHUNK OF MEMORY INTO A LARGE-ENOUGH HOLE.



→With each deallocation request, the freed chunks of memory are added back to the pool of free space



→It has been found empirically that a good strategy for minimizing fragmentation for real-life programs is to allocate the requested memory in the smallest available hole that is large enough.

best-fit

first-fit

→To implement best-fit placement more efficiently, we can separate free space into bins, according to their sizes.

For example:
→ the Lea memory manager, used in the GNU C compiler gcc, aligns all chunks to 8-byte boundaries. There is a bin for every multiple of 8-byte chunks from 16 bytes to 512 bytes.

→There is always a chunk of free space that can be extended by requesting more pages from the operating system. Called the wilderness chunk, this chunk is treated by Lea as the largest-sized bin because of its extensibility.

Managing and Coalescing Free Space :

→When an object is deallocated manually, the memory manager must make its chunk free, so it can be allocated again. In some circumstances, it may also be possible to combine (coalesce) that chunk with adjacent chunks of the heap, to form a larger chunk

→ A simple allocation/deallocation scheme is to keep a bitmap, with one bit for each chunk in the bin. A 1 indicates the chunk is occupied; 0 indicates it is free.

→When a chunk is deallocated, we change its 1 to a 0. When we need to allocate a chunk, we find any chunk with a 0 bit, change that bit to a 1 ,and use the corresponding chunk.

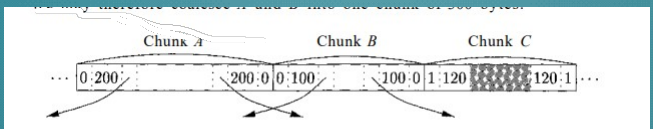


Figure 7.17: Part of a heap and a doubly linked free list

There are two data structures that are useful to support coalescing of adjacent free blocks:

→ Boundary Tags. At both the low and high ends of each chunk, whether free or allocated, we keep vital information. At both ends, we keep a free/used bit that tells whether or not the block is currently allocated (used) or available (free)

→ A Doubly Linked, Embedded Free List. The free chunks (but not the allocated chunks) are also linked in a doubly linked list. The pointers for this list are within the blocks themselves, say adjacent to the boundary tags at either end. Thus, no additional space is needed for the free list, although its existence does place a lower bound on how small chunks can get; they must accommodate two boundary tags and two pointers, even if the object is a single byte.

Manual Deallocation Requests:

→Problems with Manual Deallocation

1.Manual memory management can lead to errors, including memory leaks and dangling pointers.

2.Memory leaks occur when memory is not deallocated and can slow down programs over time.

3.Dangling pointers occur when memory that has been deallocated is accessed, leading to unpredictable program behavior.

4.Automatic garbage collection can help prevent memory leaks, but cannot prevent dangling pointers.

5.Programmers must be careful when manually deallocating memory to avoid leaving behind dangling pointers.

6.Accessing illegal memory addresses can cause program crashes and produce incorrect results, so input validation and bounds-checking are important.

7.Long-running or nonstop programs like operating systems or server code must be particularly careful about memory management to prevent issues from accumulating over time and causing program failures.

8.Careful memory management and avoiding errors like dangling pointers are especially important in applications where security is a concern, as vulnerabilities in memory management can be exploited by attackers to take control of the program or access sensitive data.

Compiler , Lexical analysis, Tokenization, Parsing, Syntax analysis, Abstract syntax tree,

Semantic analysis, Symbol table, Type checking, Intermediate code, Optimization, Code generation, Code optimization, Control flow analysis, Data flow analysis, Loop optimization,

Register allocation, Constant folding, Dead code elimination, Static single assignment form (SSA),

Just-in-time (JIT) compilation, Garbage collection, Code generation for object-oriented languages,

Virtual machine, Dynamic linking, Dynamic loading, Linker, Loader, Preprocessor, Macro expansion, Conditional compilation, Recursive descent parsing, Bottom-up parsing, LR parsing, LL parsing, Shift-reduce parsing, Parsing table, Backus-Naur Form (BNF), Extended Backus-Naur Form (EBNF), Compiler front-end, Compiler back-end, Target machine, Assembly language, Code optimization techniques, Loop unrolling, Inlining, Interprocedural analysis, Loop interchange, Loop fusion, Function call optimization, Control flow graph, Data dependency analysis, Register renaming, Static analysis.