Memory Management in the Heap

* programming languages support allocation of memory from a pool called the heap
* unlike stack-allocated memory, heap memory lives on beyond scope in which it was allocated
  + memory management scheme is needed to ensure pool is used efficiently
* memory allocator needs to:
  + find suitably large free block of mem and mark it as allocated for allocations
  + update its internal data structure to the deallocated block as free for deallocations
* memory allocator must use some kind of data structure to remember how mem was parceled and allocated
  + data structure can only go into mem so we have 3 options:
    - reserve some portion of mem for allocator itself and predicting how much mem is needed to manage the rest of mem
    - interleaving its own data structures among the data the user has allocated
    - combination of both methods
* actual program is usually called the mutator

Free List Algorithm

* allocator begins with a linked link containing one node on the heap that rep the entire pool of free mem
  + as parts of the heap are allocated/deallocated, linked list is updated to keep track of which parts of the heap are free
* e.g. assume we have 1024 bytes heap which is all free and the addresses in this block are 0x100 to 0x4FF (diagrams are not to scale and all blocks are exactly 4 bytes, which is 1 word)

Table

Description automatically generated

* + free pointer is pointer to start of first free block
    - we use global mem to store head of free list but can also reserve special space within the heap
  + first word in block stores size
  + second word in block is pointer to next free block
    - we use 0x0 to rep the null pointer but 0x1, as used in WLP4, is also acceptable
  + suppose program requests 16 bytes of mem, so we allocated 20 bytes in the heap

Table

Description automatically generated

* + - allocator looks through free list for first block that’s at least 20 bytes
    - split the block into 2, one of size 20 and one of size 1004
    - return address 0x104 to program as start of mem allocated
    - change free list head pointer to new start of free heap block
    - first word in free list is updated by subtracting amount of mem that was just allocated (i.e. 1024 – 20 = 1004)
    - second word in free list is updated to null 0x0 since free list still only has one free block
  + suppose block A is freedA picture containing graphical user interface

    Description automatically generated
    - program has been sent address 0x104 as starting point for block A because we don’t want the size to be mutated
    - when allocator receives request to deallocate block at address ptr, must use address ptr – 4
    - allocator inserts freed blocks in increasing order of addresses so this newly freed block is added to front of free list
    - free list head pointer now points at 0x100
    - second word in block A is updated to be a pointer to previous free list head pointer
  + suppose block B is freed

A picture containing graphical user interface

Description automatically generated

* + - allocator traverses nodes of free list and determines that address of block B is between block A and big free block
    - second word of block B is updated to point to the previous next pointer of block A
    - second word of block A is updated to point to block B’s address
  + we sort in increasing address order so allocator can more easily determine when to merge 2 adjacent blocks into a single block of free mem
    - after inserting new free block, allocator might merge 2 blocks
    - update size of first block to sum of 2 blocks
    - update next pointer of first block to point to next pointer of second block
    - merging process can be handled with 2 explicit checks and doesn’t require recursion
* if requested mem can’t be allocated (i.e. no large enough free block or free list is empty), common convention is to return null pointer
* one edge case to consider is when requested space is exactly 8 bytes smaller than the largest free block
  + e.g. user requests 24 bytes and there’s a free block of 32 bytes
    - remaining 4 bytes of the split free block isn’t enough to store both the size of the free block and address of next free block
    - solution is to allocate entire 32-byte block
* common problem with heap allocation algorithms is repeated allocation and deallocation of mem can create holes in heap
  + e.g.

A picture containing chart

Description automatically generated

* + - last request is to allocate 20 bytes of mem and although there’s 15 bytes available, it’s not big enough
  + repeated allocation/deallocation can cause fragmentation (i.e. required bytes of mem are available but they’re not available in a contiguous block so they can’t be allocated)
* there are a number of heuristics-based approaches that attempt to reduce fragmentation
  + first fit: find first block that can satisfy request
  + best fit: find block that leaves smallest/no hole behind
    - not necessarily the best because it’s prone to creating the smallest and least reusable holes
  + worst fit: find biggest block and allocated from that

Binary Buddy System

* binary buddy system restricts itself to only allocating blocks of size 2k for some k
  + if block size is not available, smallest block bigger than 2k is split into 2 buddies of the same size
* free blocks list and allocated blocks list is kept
* e.g. start with a heap of 1024 bytes and the program requests 20 bytes (we need 24 bytes)

Table

Description automatically generated

Table

Description automatically generated

* each block is assigned a code in the system
  + biggest block gets code 1
  + if we split block, then left buddy gets code 10 and right buddy gets code 11
  + a block’s code is stored as bookkeeping info in first word of the block
* 3 notes on a block’s code:
  + block can find its buddy by flipping its own last bit
  + if a block’s code has n digits, the size of the block is 1024 / 2n−1
  + since first bit is always 1, we can store it using more space than needed so we can tell how long the code is by counting number of 0s before first 1
* can determine starting address based on block’s code
* when mem is allocated, free list can be searched for appropriately sized block by find correct number of digits in block’s code
  + e.g. to allocate a 32-byte block we need to look for a block which has 6 digits in its code since 1024 / 26−1 is 32
  + if free list doesn’t have the correct size, choose the next largest number of digits in block’s code and split it
* when block is deallocated, search for its buddy in the free list and if found, merge them
  + repeat process until all possible merges are done
* disadvantage of using buddy block system is that it causes internal fragmentation

Implicit Memory Management: Garbage Collection

* many languages, such as Java and Racket, perform automatic garbage collection, which means they automatically deallocate mem that was allocated by program once that mem is no longer needed
  + e.g.

Text

Description automatically generated

* key to automatic garbage collection is determining which blocks of mem are no longer going to be used by the program
* reference counting algorithm keeps track of number of pointers that point to each block
  + from the time a block is allocated, algorithm must watch each pointer update so that reference count to each block of mem is kept accurate
  + whenever reference count of block reaches 0, block of mem can be deallocated
  + one limitation is when there’s circular references, blocks would never be deallocated
    - e.g. one block refers to another block which refers back to the first block so the reference counts for both are never 0 even if they aren’t accessible from anywhere else in program
* mark and sweep algorithm begins with mark phase where it scans the entire stack and all global vars for pointers leading into the heap, then marks all heap blocks as reachable
  + if any marked blocks contain pointers, the algorithm repeatedly follows them all to discover new parts of the heap that are reachable
  + once entire reachable part of heap has been marked, sweep phase starts and any unmarked block is deallocated
  + this algorithm belongs to class of “Stop the World” algorithms since when the garbage collector runs, program is stopped so it can’t make any changes to mem
  + disadvantage is that although it can accurately collect garbage, it has to stop program from executing every time collector runs
* copying collector algorithm splits heap into 2 halves called from and to
  + mem is only allocated to from part of heap
  + when from part is filled up, garbage collector runs and copied only the reachable parts of from to to, then reverses roles of the 2 halves
  + one advantage is automatic compaction so it can be laid out contiguously without any fragmentation
  + disadvantages include:
    - it’s another “Stop the World” algorithm so it’s not suitable for applications that expect real-time responses
    - available heap mem is halved
* copying collectors tend to work well when few objects survive collection
* mark-and-sweep collectors work better when most objects survive collection
* generational garbage collection classifies objects into different generations based on when they die, and fuse multiple techniques
  + new objects allocated in youngest generation and collected through copying collection
  + objects that survive first few collections are moved to older generation which uses mark-and-sweep algorithm