HEAP MANAGEMENT

Heap is a virtually continuous portion of the process's address space and is entirely Software managed. . It usually contains a starting point, limit mentioned by the system and an end point called the **break**. The break marks the end of the mapped memory space, that is, the part of the virtual address space that has correspondence into real memory
brk – Places the break point at a given address. Returns 0 if successful, -1 otherwise.
Sbrk – Move the break by given increment. If increment is 0, returns the current break point's location.

Linked list of free nodes. Always keeps the address of 1st free node.
In the 1st 8 to 4 bytes, keeps track of how big that node is, pointer to the next free node. This has to be done generically. They don't do any error checking in-order to be very fast.

NAIVE IMPLEMENTATION

SETUP:
It is started a big free blob of memory. The 1st 4 bytes which is the header, has NULL inside it. It means, there is no node following this one. It also has information about how big the size is.

MALLOC:
A very naive implementation of heap management consists of viewing the entire heap as a singly linked list of blocks. Each block has a **header** which contains **meta-data** about the data block. The typical contents of the meta-data are

1. **Size of the block**
2. **Free or not**
3. **Pointer to the next block in the linked list**

Whenever a request is made, this master linked list is traversed, and a first available block is chosen. This block is marked as NOT free. To avoid fragmentation, if the size of the block is much greater than the requested block, then the block is split into two blocks. One block whose size is the size of the header + requested size and other, whose size is the remaining size. The next block is marked as free

SCANNING FOR FREE BLOCKS:
1. FIRST FIT
Overlay a LinkedList of blobs. It will use the 1st 4 bytes of unused blobs, to store the address of the next free blob. It can also have a circular linked list.

2. BEST FIT
Scan the entire LL so that, as little as extra memory is avaliable in the free nodes.

3. WORST FIT

4. Optimisation
They remember the last place where the served for the previous request. So the next search can start from this place and thus saving time.

FREE:
NAIVE FREE:
The major problem with malloc is fragmentation. A very naive free, just turns on the free bit. When we do a free, none of the contents are changed. Only the 1st 4 bytes are actually modified and they are added in the linked list. It would not go and clear the information. It could, but is a time consuming operation. In theory it can be 1MB. So a waste of time.

INTELLIGENT FREE
If the pointer is valid:
– we get the block address
– we mark it free
– if the previous exists and is free, we step backward in the block list and fusion the two blocks.
– we also try fusion with then next block
– if we're the last block we release memory.

– if there’s no more block, we go back the original state (set base to NULL)

If the pointer is not valid, we silently do nothing.

REALLOC:
NAIVE REALLOC

A very naive realloc contains

1. Allocate a new block of the given size using malloc
2. Copy the data from old to new block
3. Free the old block
4. Return the new pointer

INTELLIGENT REALLOC

1. If the size doesnt change or the extra available size is sufficient, we do nothing.
2. If the next block is free and has enough space, we fusion it with the current block and split it if necessary

BETTER IMPLEMENTATION

The haap can be managed better by having **several linked lists**. One for the normal allocated linked list. There can be other separate linked lists. One for size in range 0 to 64, other for size in range 64 to 256... like that. Thus, after aligning the input size with the size of the header, the apropriate list is only searched. Eg, if the size is 20, then the list containing free blocks of size 64 is searched and a block is allocated from there. When a corresponding block is freed, it is added to the head of such a list. Thus insertion operation is O(1) Some implementation, will just remember the free calls and will not only commit to the heap during the next malloc call since it could save a lot of time.

PROBLEM WITH FREE

1. **Free does not check** if it is a valid malloc'd memory

When a pointer is passed to realloc or free, it assumes it is a legitimately assigned memory and blindly backs up 4 bytes and thinks this is header.

If the code is

```
int main(){
        int arr[100];
        free(arr);
}
```

Free will take the base address of this arr array, back up 4 bytes and try to incorporate this into heap.

HEAP COMPACTION:

1. Can we re-arrange the free bytes to avoid fragmentation?

Compacting the heap. Wat remains is now assumed to free. But there is a problem here. The moved changes are not reflected to the client. The client is still pointing to the old place. This is problematic implementations. MAC did compacting heap.

They **clip of some part** of the heap. The larger portion is used in a traditional way by the heap manager. **The top small portion is managed as handlers**. They hand out **handles**, which are not single pointers to data, but **double pointers**. Instead of handing out pointers, they hand out double pointers and maintain a list of single pointers. If we ask 80 bytes via handles as opposed to a pointer, it could maintain a table of master pointers and hand out the address of that to the client. The client is now two hops away from the data. But the, advantage is that the table is owned by the heap manager.

```
void **handle = newHandle(40);
```

We can put a handle lock and say the OS do not change the handles in the table.
When done, we call handle unlock.

If we over-write the array, then it might go and write into the next allocation's header.