CS280 Notes 1/5 “Memory Manager”

by Christian Sagel

DigiPen is starting an universal note taker program.
They can have somebody who takes notes everyday in class and posts those notes.
Now those notes will be global rather than that just for DSS department.
The information is in an application and contract form.

Fragmentation Control

Like disk fragmentation, there’s memory fragmentation.
When you defrag it moves everything next to each other so its contiguous.
When you allocate memory, due to the algorithms for fitting there’s fragmentation present.
Requires handles: pointers to pointers.
Usually seen in managed languages where you don’t deal with raw memory.
We previously did variable size in CS180. For CS280 we will be doing fixed size.
It is more performant for them to be fixed size.
Coalescing (merging adjacent free blocks)

Handles: pointer to a pointer. Double indirection. We can move the raw pointers and shift memory around/coalesce. This is how GC languages work. This takes time, though.
Normally GC only happens when you need it. When you start freeing things when you need to reuse them, that's when you start getting fragmentation.

There are third-party add-ons to the C++ compiler that will add garbage collection (!)

Debugging Capabilities

Audit memory usage, consistency checking, initialize blocks to certain values. We can detect errors.
Invalid memory block detection. Memory leak checking.
The MM we are learning to implement will be quite useful for game performance.

(We were given 2 handouts for the assignment)

One of the best things when getting started with technical things, is to do diagrams.
Do not hesitate to ask questions!
Mead almost missed the sign-up sheet?

Assignment

The diagrams are currently 32-bit, though we will be working almost exclusively on 64-bit.

For MSVC we use long long. It is supported in C++11;

The config struct has 6-7 fields that control whether there's debug on, the padding, etc.
If you have 5 or 6 more parameters you probably forgot to add some.

There's a GenericObject strut that only has a pointer to the next GenericOject.
Because we are dealing with void pointers, raw memory. We want to cast stuff ?

When we first call the OA constructor, we pass the size of the struct.
Each page is a linked list of pages.
We have 2 important pieces of information: the page list and the free list.

The free list is a singly linked list of all the blocks of same size. If they are on the free list they are available to be given to the client.
The page list is the memory blocks that we get from the operating system.

On Monday we will talk more of advanced features.
We should be able to add and remove blocks from the list.
When we start the page list is currently pointing to null. The free list is also pointing to NULL.
If the freelist is empty/NULL there’s no memory available.
After we allocate our first page, we are going to set the page list to point at the first pae.
The first 4 bytes are the next ptr. Because its our first page, its ‘next’ pointer will be still NULL.
(On 64-bit systems, it will be 8 bytes)

The underlying data structure is a simple singly-linked list.
Now we have to take each of the 16-byte blocks and put them in the free list.
On a normal SLL we would call new 4 times.
But because the memory is already available and ready for use, we can skip allocation.
We can now go to the second phase and move these blocks to the free list.
We are going to take the free list and point at the first node in the array.

We are going to hijack the first 4 bytes of the node.
A linked list of linked lists.
We are confused because normally we go right to left.
To find the address of the next block, “next += sizeof(object)”
The freelist is a char pointer, but it's being casted into a generic object.
Do not make freelist/pagelist into void pointers.

Have a temp pointer. Point temp to the current node.

Add sizeof(object) to the pointer and now the freelist will point to the next node.

We keep adding nodes to the freelist in this way.
The driver is going to dump the linked list back to front, right to left. This is why we can’t go left to right.

Page 5 is what we see after calling the constructor.
Allocate: We remove the head of the freelist, linked list.
temp = freelist;
freelist = freelist->next;
return temp;

Free
Pts a block onto a SLL
Check whether it there
Check the address if it's valid

Allocate/Free are only dealing with the freelist currently
Pagelist is internal, Freelist is external and seen by the client?

We still have those next pointers there, because they are still being tracked by the memory manager. The user is free to scribble over that memory and hose himself.

Free: Puts a node on the front of the freelist, a linked list.

Both of them are 3 instructions, barring other features.

The biggest bug in the internet is scribbling over the stack and hosing the return address.

Now that the client owns those bytes, they will overwrite the next pointer.
When freelist finally points to NULL and the user requests for more memory, we request another block of memory for the page list. The new block will point at the last block.

If the user tried to call allocate after both pages are full, we would throw an exception.
If the max pages are full, we keep going until the OS is at its limit.
When we want to free the first block, we just set the first 4 bytes to NULL;

In the freelist the blocks don’t have to be contiguous, and most likely won’t be. This is how real

MMs work.

char* next[4]

Pagelist can be a genericObject or a char*.
We cast the address to a generic object pointer,
Let’s say we had a GO* g;
g->next = pagelist;
GO* will fill the 4 bytes at the beginning of block.

Any kind of a linked-list structure benefits from having a memory manager.

Mead cannot stress enough: helper functions!

allocate/free end up calling 8/9 helper functions.
The more helper functions we have the easier it is to understand our code.
By next week: constructor, destructor, allocate, free

CS280 Notes 1/11 “Memory Manager”

by Christian Sagel

Questions
When allocating pages, do we set the next pointer on each ‘block’ to point to the next block before it gets casted?
Us vs the client in the memory manager. When we first allocate that page, we need to set the bytes to a specific signature if debug is turned on
Use ‘memset’ to set the whole thing to ‘AA’
When the OA gives the block to the client, we are gonna mark them as ‘BB’
When we give it back to the OA, we mark it as ‘CC’. This means that we are still using a block that we freed.

Padding

The ‘DD’ pad bytes signature. Say we had a 16-byte block that we allocate. On debug mode these bytes are turned on. When pad bytes are set, we set pad bytes at the start of the block and at the end of the block. (Of same size). The signature for these pad bytes is ‘DD’.
Padding is not shared between blocks. Each block has its own bytes of padding..

We can memset these pad bytes (Takes an address and how many bytes to set)
We put pad bytes with debugging on.

Alignment
The alignment bytes. Depending on the size of the block and how many bytes you want to add:
We originally allocated a page and memset them to AA. We took 8 bytes and set them to 0.
Say we wanted 8-byte alignment, the address at the block where it starts from needs to be divisible by 8. I have to add whatever number of bytes needed in order to make the address divisible by the size specified. Because the first block has an extra 4 bytes because its the header,
Alignment: 4/0 (left/interblock) (the first block/between blocks)
Once given the config of the padding, alignment we have all the information needed. We have the size of the pointer, size of the objects to calculate the value.
In RL, the alignment would be decided on the architecture of the machine.

Header blocks
Padding bytes, alignment bytes and the 5 bytes used by the header block.
The rightmost bit of the header block is 0/1. 0 if free, 1 if in use.
The linear way to check whether the block is being used is by doing pointer arithmetic depending on the address relative to the header block. IF we dont have a header, we would have to walk the list. Do mod arithmetic to check for boundary.

```
if (headersblock on)
        look at bits();
else
        walk the free list()

        Do it in order:
```

1. Check whether its in one of the pages
2. Check whether its on the free list

We calculate the amount of bytes needed in order to make sure every block is aligned.
We figure out the alignment of the first block and the alignment of every block after.

Header block modes:

- None: Duh
- Basic: Every header is 5 bytes. The first 4 bytes are the allocation number, the 5th right-most byte is a flags byte. The OA keeps an internal allocation counter that it hands out to each allocated block.
- Extended: A 2-byte use counter and an user-defined field of any number of bytes.

- External: The header itself is a pointer to a chunk of memory outside the block itself. An user-defined struct of indefinite size. This will be done with normal allocation, not with a memory manager of its own (lol). Layout is affected by the size of the pointer.

If the client gives you a label but you don't have external turned on, ignore it.

We have all the information required to allocate the page, modify the fields for signatures, padding bytes, alignment bytes.
We can think of the address we get from the OS as starting from 0.
The XX stand for the addresses.
If the max number of pages has been reached, throw an exception.

A lot of the checks are done on FREE. Check that blocks haven't been corrupted by checking signatures padding bytes.
When comparing pointers, we can check equal and unequal:
We should be able to do a 1-line expression in constant-time to check whether the address is in the page.
First check whether the address is in the page.
if (block_on_address)
Second check whether its in a boundary in the page.
if (block_at_boundary)

Turn addresses on while debugging, off when trying to diff output.
SHOWADDRESS