Writing a memory manager

From OSDev Wiki

This page is a work in progress and may thus be incomplete. Its content may be changed in the near future.

Difficulty level

Medium

Contents

First fit MM

It's not that hard to implement a basic first fit MM. By memory manager, I don't mean Paging (you just keep a list of free/used pages), but rather a simple library you can use in userspace and in your kernel (or globally if you have a flat memory model).

What I'm going to talk about assumes you know where the chunk of free memory is you can play with. For a flat memory model, this can be from the end of your kernel code (say the 1/2MB mark) to the end of the system memory (GRUB can give you this info). For a paging kernel, this can be the beginning of the page to the end of the page.

malloc()

Okay, the first step is to get out a pen and paper - I don't mean a keyboard and text editor since you can't draw diagrams (a stylus/tablet is okay with your favourite-graphics-program/OneNote/Windows-Journal)..

Anyway, my point is.. Do you understand the basic concept? Sketch how the memory will be laid out on paper.. Don't start writing your allocate function straight away.. That should be one of the last things you should do.. Actually design how you want your memory to be managed, where you want to store the information (in a header before, in a global bitmap, etc).. What do you want to store in the header/bitmap? If it's a header, what sort of data will be saved in it (pointer to the next free space, if it's free or used)..

Remember this is your operating system! Plan it how you would like..

The important thing is to start from an OVERALL view.. Not sitting at a keyboard with a blank malloc function expecting to know what do do..

Then write a basic structure for the header/bitmap..

When you know how memory is stored, then you can go on to your allocation function. Close that text editor, we're not done with that pen yet :) Again, start with a top down view. What, exactly, does a memory allocator do?

1. Find free block
2. Mark it as used
3. Return its address

Then break it up..

How do we find a free block? (this is for a first fit method with a header, but for a basic buddy system all you need is some imagination)

- Start with a pointer to the beginning of the free memory
- Does this header have an End Of Memory status?
  - Yes
    - Allocate a new page

      or

    - return an error (such as a pointer to 0)
- Does this header say this memory is free
  - Yes
    - Is (address of next header - (address of this header - size of header)) larger than the size wanted
      - Yes?
        - WE FOUND A FREE BLOCK!!!!
      - No?
        - Point to the next block and go back to step 2
  - No
    - Point to the next block and go back to step 2

Once you have found one, set it's used bit to used. If the space found is significantly larger than the space needed, then it is generally a good idea to split it up (cut it in half, then half again, or add another header right after the used space and update the pointers.. use your imagination!) ;] Then return the address!

free()

Don't program it yet! Now you need to write your free function.. You need to basically go through the same process.. You need a way to find out where about in your bitmap or which header is associated with the address you're given. With a header it's simply address - size of header. Set it's bit to free.

The next step is to scan through the memory and merge free blocks (e.g. if next pointer's header is free, search the next pointer's header, then the next, until you find a used block, then set the current block's next pointer to that used block, skipping over the free blocks - if any of that made sense :D).

Just keep breaking it into smaller, and smaller blocks.. Make sure you understand 100% how each process works.. The key is to understand it fully before you start coding! Write down every equation for everything. It is essential you know how your code works, and have it in readable form on paper, so when an error occurs it's easy to debug.

Testing

Only then, you can start coding. Once it's written, you should try it out in a simple sandbox environment.. Such as allocate around about 5 random sizes, deallocate a few from the middle, re allocate them, and get them to print out all their memory addresses at each step, then take the time to work it out on paper to see if it's doing it correctly. The main problem I've had is with the freeing (combining free blocks together) which I had to print out every step to the screen to find the bug (I forget what it was, a = instead of a == I think).

Don't copy someone else's code! You can copy their basic theory, but plan how you want it to work in YOUR OS on paper, and implement it YOUR way. That way, you know exactly how it works, and you can debug it and extend on it yourself.

Good luck with your MM. :]

Best fit MM

A basic idea of a best fit memory manager is same as a first fit one. Having blocks in memory. But the problem with first fit is the fragmentation which occurs when splitting blocks way bigger than we need, when there are smaller fitting blocks.

Keeping track of free blocks

You need to know where each free block is in memory, and how big it is. You can store this in an array, or binary tree, or whatever. The way you sort the list is important. If we sort the list on size, small to big, we can look for free block starting at first entry, and as long as the current node is not big enough, going to next node. This way you will find the best fitting block. If no block is found, you need to ask the virtual memory manager for more memory.

For making this work you can make some nice functions. Those functions (add for example) finds the best place to put a node (free block), and then moves all other nodes a bit for making place for new node. Removing a block is easier. Just find the given block in the list, remove it and move all other blocks to fill up the hole.

Merging and splitting blocks

Merging and splitting free blocks becomes a bit more work because you have to keep your free block list updated.

Good luck making a best fit memory allocator!

- This page has been accessed 51,515 times.