

LZSCC.211 – Operating Systems – Coursework 2

This is the final assessed exercise for 211 and carries 20% of the module marks. As the last exercise in a coursework heavy module, you will likely find it more challenging, and starting early is really important. The exercise is different in that it is a design and implementation task – you must develop a solution to a real-world problem given a ‘blank sheet of paper’.

Overview

The task is to implement a memory allocator for use by application programs. This must be usable as a replacement for the existing `calloc()` and `free()` functions, and conform to the requirements given below. Don’t try to understand everything or implement everything at once, it will take time.

Starting Point

Look at the lecture material for session 2, particularly that relating to memory allocation. Some of this is linked, along with some hints on C, within the Moodle section covering this exercise. You should also look at the Linux documentation for `calloc()` and `sbrk()`, by using the `man` command in a console window, or by looking at the many copies around the web, examples being:

`man calloc` on Linux, or <https://linux.die.net/man/3/calloc>

`man sbrk` on Linux, or <https://linux.die.net/man/2/sbrk>

Web copies of the Linux manual pages are also available on many other sites.

Requirements

Your solution must be implemented in C and operate as follows:

- Follow the existing calling convention, and be callable as per:
 - `void * new_malloc(size_t nmem, size_t size);`
 - `void new_free(void *ptr);`
- Users request memory by calling `new_malloc()` and return it with `new_free()`
- User requests for memory must be served from memory held by your implementation
- You must always allocate existing free memory in preference to claiming new memory from the system
- If, and only if, your system has insufficient memory to serve a request, your code should claim an additional 8KiB (8192 bytes) of memory using a call to `sbrk()`
- You cannot assume your code is the only thing calling `sbrk()`, i.e., there could be gaps between the memory returned by successive calls to `sbrk()`
- You should maintain at least one free list of memory available for allocation

- Pointers must be within the free memory, not a separate data structure
- `new_malloc()` should always split the smallest free memory region of sufficient size, return any excess memory to the appropriate free list, and return the requested number of bytes to the application
 - If there are multiple regions of the same size, the first on the free list should be used
 - There is a minimum size for a free region `malloc()` can handle, depending on the amount of metadata needed to manage each free region. If the excess is too small, it can be returned to the application along with the requested memory

`new_free` should add returned memory at the head of the appropriate free list

Operation

You must provide a program to test/ demonstrate your implementation. This program must repeatedly take console input of the form `A<number>,<size>` or `F<addr>`, where `<number>` is a number of entries to allocate, `<size>` the size of each entry, and `<addr>` is a pointer/ address for a region to free, as returned by your `new_malloc()` function. For example, if `A100,2` returned `0x459a2`, `F0x459a2` would return the memory to the free list. Allocation must display the address returned by your `new_malloc()`.

After each input/ operation, your code must display the amount of free memory, i.e., the sum of all free memory on all free list(s), and then the address and size of each region on each free list, for example, with three entries on a single free list, of size 100, 75, and 25 bytes, the output must be in the following format:

Total memory: 200 bytes

0x1a924-100 | 0x1b046-75 | 0x1b205-25

Assuming the free regions start at addresses `0x1a924`, `0x1b046`, and `0x1b205`.

If you get to the point of having multiple free lists, you must prefix each list with the size held by each the list, for example:

Total memory: 5181 bytes

1024: 0x1a924-1105 | 0x1b046-1200 | 0x1b205-1096

512: 0x1a924-576 | 0x1b046-580 | 0x1b205-624

Line breaks can be added as needed, but please use print formatting to make your output as readable as possible - it is in your interest that markers are able to easy follow what is going on.

Submission

You must submit your solution to Moodle by the posted deadline, You should submit one zip file containing your solution and a short report (not longer than 3 pages): the report is a chance to explain your design ideas, how you tested your code, any issue you faced, or any potentially interesting point arisen during your work.

Code and Commenting Expectations

Code should be maintainable – it should not be overly verbose, or so tightly crafted that its operation is unclear. You should also avoid unnecessary repetition of code/ work. Code should make good use of language, i.e. appropriate use of structures, functions, loops, etc. Code should be properly commented, with the level of commenting uniform throughout all files.

Comments should clearly outline what each logical set of statements aim to achieve. Over commenting, where comments just state the obvious effect of statements, or are such that minor edits to statements are likely to require changes to the comments – likely to lead to them diverging from the code over time – should be avoided.

The purpose of variables should be clear from naming or a comment at their declaration. Avoid the use of overly long variable names.

Getting Going

Start straight away... don't wait before you start working on the problem or you'll run out of time. Importantly, make good use of your timetabled lab sessions – don't waste valuable support time by not preparing beforehand/ getting on with coding outside of the labs.

This may be one of the first times you've had to start with a blank sheet of paper and develop a complete solution to a problem. As such it may seem far more intimidating than it is. Some things you might try to build your confidence...

Firstly, if you're unsure of something in C, or how a data structure works, write a small program to experiment, and play with it until you understand and are confident. Don't try tackling this at the same time as worrying how it needs to fit in your main program.

Always go step-by-step, and properly test each step as you go – what should the code you've just written do, what are the edge cases that might cause it to fail? Write a few lines of code to properly test each step you write and use print statements to output results so you can confirm everything is working as expected. Have one or more debug flags or use `#ifdef` to turn your print code on or off when it starts getting too much. Ideally you'd run all your tests regularly, but you always need to properly test any code you've just added.

If you jump ahead without proper testing, which can be tempting, the risk is that when things go wrong you don't know whether it's in the code you've just written, or whether the fault is in something you wrote a few days before and can't quite remember how it works.

Testing as you go really narrows down what can go wrong and what you need to test, and can save hours of head scratching later. Proper testing is good practice for all coding but is essential with systems code as small mistakes can be the source of significant problems that are really hard to find and debug.

Again, always test edge cases, and what happens if your code gets some unexpected values passed to it. When you come to using C pointers to create an explicit free list, as opposed to just scanning every header to find unused space, make sure you understand and can get a simple linked list working before you start – there are plenty of examples on the web. Try this in a separate program

using the regular `calloc`. Remember that for your new `_calloc`, the pointers must be within the memory you are managing, i.e., that returned by `sbrk()`.

Write some debug code that outputs the content of the list. Double check you can add and remove things – don't forget to check what happens when you start or end with an empty list.

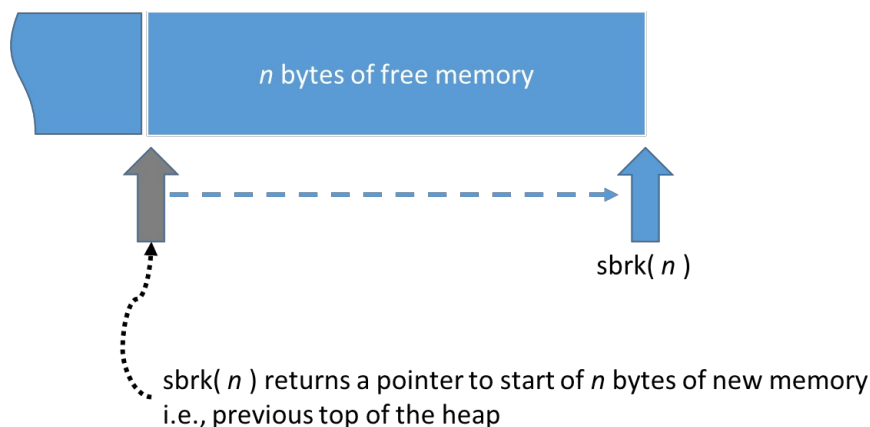
Allocate a large area of memory for a set of structures, and check you can use pointers to move between them, picking out particular fields in each.

The Moodle page has links to some examples showing the use of pointers, structures, and pointer arithmetic. If you're unsure, these may be worth a look.

Step by Step

Traditional implementations of `calloc` use `sbrk()` to extend the heap and get memory from the OS.

A good starting point is to ensure you understand what this does and how it works.



Whenever `calloc` needs memory it requests a fixed size chunk of memory, in our case always 8KiB or 8182 bytes, using `sbrk()` and then manages this memory, handing out small runs of bytes in response to application calls to `calloc()`. Each application has its own address space, and thus its own heap, and its own instance of `calloc`. There could be multiple threads, and `calloc` may not be the only thing using `sbrk` to obtain memory.

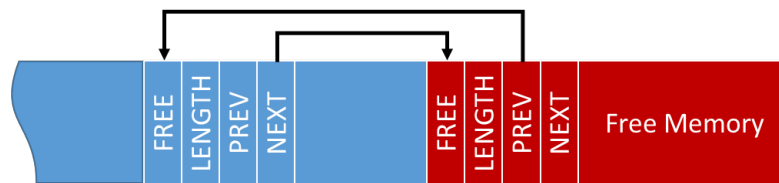
`Calloc` includes metadata in the memory it manages. It needs this to know the size of each area, and whether it is currently used (allocated to the application), or free. Remember, the application should return memory to `calloc` using `free()` so you cannot assume there is a single used region followed by some free memory.



This can be improved upon... Having a doubly-linked list makes it easier to scan forward and backward for free space, and allows for coalescing of adjacent free areas to reduce fragmentation. This can be taken further by having separate doubly-linked lists for free areas of different size,

LZSCC.211: Operating Systems

typically falling within powers of two, for example, up to 64 bytes, 64-127 bytes, 128-255 bytes, etc.



Notice the Prev and Next pointers are only needed for areas that are free, i.e., on a free list. This means memory occupied by these pointers can be given to the application as part of the memory returned in response to its requests – with many small requests, this can save a lot of memory.