SCC.211 Operating Systems Exercise

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 malloc( ) and free( ) functions, and conform to the requirements given below. This is a multi-week exercise and you'll need to build up to a final solution over a few weeks – don't try to understand everything or implement everything at once, it'll 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 malloc( ) and sbrk( ), by using the man command in a console window, or by looking at the many copies around the web, examples being:

- `man malloc` on Linux, or https://andrew-scott.uk/docs/man/man3/malloc.3.shtml
- `man sbrk` on Linux, or https://andrew-scott.uk/docs/man/man2/sbrk.2.shtml

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  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<bytes> or F<addr>, where <bytes> is a number of bytes to allocate, and <addr> is a pointer/ address for a region to free, as returned by your new_malloc( ) function. For example, if A100 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: 200 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.

Extras

The following build on the basic implementation and allow for a higher grade.

- Optimise the use of metadata to maximise space available by allowing applications to overwrite free list pointers, etc. that are only needed when memory is on a free list
- Implement multiple free lists to speed up searches, with each list holding regions with the same base size, e.g., 128, 512, … Each free list should be output separately as shown.
- When memory is added to the free list, your code should coalesce this memory with any free space immediately before and after.
    - You should be able to coalesce memory returned by successive calls to sbrk( ) if nothing else has called sbrk( ) in between. Think how you can use sbrk(0) to check.
    - Note this may require moving free regions to a different free list
- Check pointers returned by applications using new_free( ) are valid before attempting to add anything to the free list
    - You should also check for applications trying to free memory more than once
- Serve requests for large amounts of memory, more than 8KiB (8192 bytes), by calls to mmap( ) / MAP_ANONYMOUS
    - Such memory must be freed with munmap( ) when returned by an application calling new_free( ) – don't mix memory areas returned by mmap and sbrk
    - See `man mmap` or https://andrew-scott.uk/docs/man/man2/mmap.2.shtml

Marking

You must submit your solution to Moodle by the posted deadline **and** demonstrate your submitted solution in your assigned practical session to be graded. Solutions must be your own individual work, and you must not submit any third party code as part of your solution – please remember the plagiarism rules and penalties. Demonstrating code different to your Moodle submission will result in zero marks/ an F4 grade. **Your solution must not use the existing memory allocation functions, such as, malloc( ) or free( ) functions. The only exception are sbrk( ), mmap( ), and munmap( ), and they must be used as outlined above.**

The following are indicative grades based on key features; however, markers will be also be taking into consideration other aspects of your code and understanding when determining your grade, including commenting, code structure, use of data structures, use of C, etc.

Grade	Key expectations – these need to be completed in order
A+	As for A **and** check memory pointers returned by new_free( ) are valid, and add code to use mmap/ munmap for servicing requests for large amounts of memory (8KiB or more)
A	As for B **and** coalesce adjacent free regions when an application calls new_free( ), including adjacent regions obtained from more than one sbrk request, placing the resulting region on the correct list **and** optimise the use of metadata by allowing applications to overwrite metadata (pointers) only required for regions on a free list
B	As for C **and** implement multiple free lists, each holding free regions of a binary multiple in size, for example, ...63, 64..127, 128..511, 512..1023, 1024..2047, 2048..4095, 4096..8191, 8192…
C	As for D **and** maintains an explicit doubly-linked free list using C pointers You must be able to demonstrate multiple calls to sbrk work correctly at this level
D	Must demonstrate a working system meeting the expected requirements and operation, i.e., able to **reliably** allocate **and** free regions of memory, and produce output as required
F3	Able to allocate memory, manage memory regions correctly, and give expected output, but new_free( ) not working as expected.
F2	Able to allocate memory, but **minor** errors in way memory regions are managed.
F3	You should be able to demonstrate good understanding of problem, have made a reasonable attempt, show you have appropriate C structures defined, and have a good strategy/ plan for coding and debugging.

To gain a particular grade you must be able to demonstrate your solution works correctly and fully at the appropriate level, that it properly/ fully meets the specification, and you are able to fully explain the expectations and operation at this level. Note: explanations must go beyond reading comments in code, and we will be testing your code works with odd/ edge cases, so check before you submit.

A plus grade may be given for solutions fully meeting the expectations for a given grade that are well structured/ presented, fully/ well commented, and where a particularly good explanation is given. A minus grade may be given provided the expectations of the next lower grade are met, but there are minor failings in the solution for a particular grade, or where explanations are poor or unclear.

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.

Each function should have a header comment outlining what the function does, and any parameters or return values should be explained – think JavaDoc.

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 an 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. It's amazing how often we see code break in marking sessions when given inputs you might expect would cause a problem.

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 malloc. Remember that for your new_malloc, 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.

Things to Ponder

How would your solution work for other allocation strategies? …look at the session on memory allocation, and investigate, First Fit, Next Fit, Best Fit and Worst Fit allocation strategies.

When adding anything to a free list, should it be added at the head, the tail, or somewhere else? …what would be most efficient, and why?

Why should you avoid calling sbrk( ) until absolutely necessary?

If you can defragment a disk, why can't you defragment the memory managed by malloc( )?

…what is it about disk storage that makes defragmentation possible, and could something similar be done for memory? …if so, what constraints would there be?

Why does malloc( ) use mmap( ) in preference to sbrk( ) for large allocations?

Why might implementations of malloc be moving to the use of mmap instead of sbrk to create a 'heap', as well as using mmap in the traditional way for large requests?

If an application might be expected to make a large number of malloc( )/ free( ) calls for a small number of fixed sized structures… is there a better approach to using malloc( ), and if so what?

What would need to be done to make your implementation thread safe?

Help and Support

Questions on the exercise should be asked in the lab sessions or in the Moodle forum. Frequently Asked Questions (FAQ) may be covered in the forum on Moodle, or in some cases covered in lecture sessions. Other sessions will normally be reserved for questions or discussion of taught material.

You are expected to join your timetabled lab session, and start the exercise early – even if you're not coding, you can be thinking about the problem and possible solutions.

TAs will be in the timetabled labs to provide help and support, but please remember they are not debugging tools – they will expect you to have tried testing your code step-by-step as you build up a solution and to have gone through a process of adding print statements to see what is happening. Don't expect them to walk or 'hand-hold' you to a solution, and please don't hog them, or present loads of untested code and expect them to tell you what's wrong. Expect to be asked what you have tried, how you narrowed down the problem, and walk them through relevant debug output.

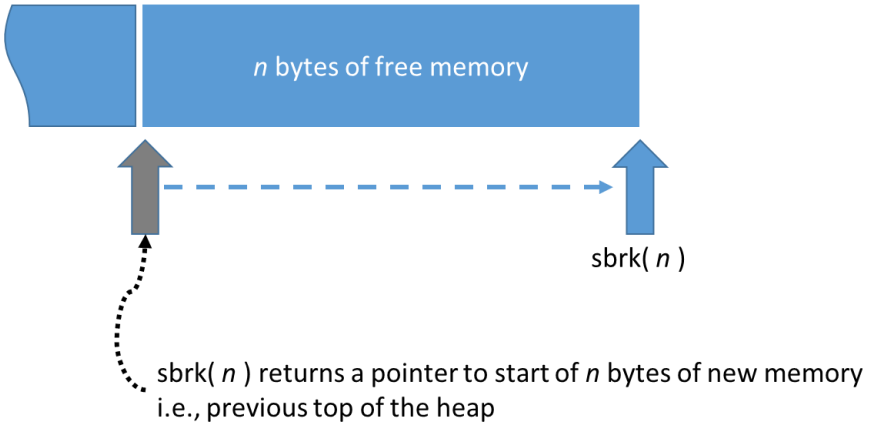
Be clear what your code should do… and what do a set of print statements show it to be doing? Does the problem change with different input? If so, can you use this to narrow down the cause?

The best use of TAs is to seek explanations of what the system should be doing, and to bounce ideas off them – whether design ideas, or ideas for debugging strategies.

Having said that, the TAs are keen to help, so don't feel lost. The sooner you ask for help the easier and more effective the help will be.

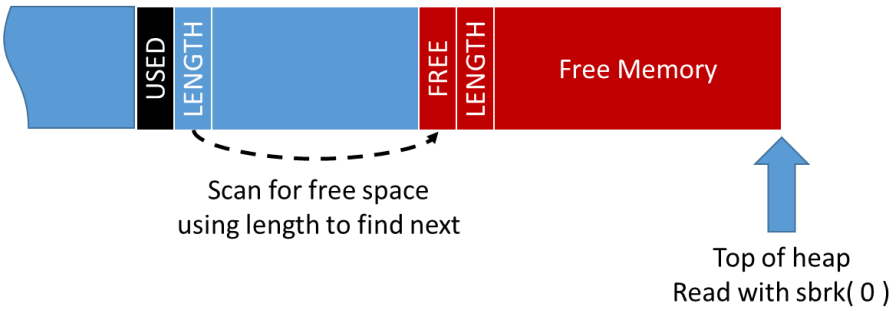
Step by Step

Traditional implementations of malloc 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.

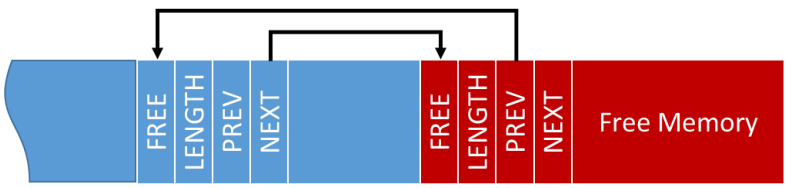


Whenever malloc needs memory it requests a fixed size chunk of memory, in our case always 8KiB or 8182 bytes, using sbrk( ) and then manages this memory, handing out small runs of bytes in response to application calls to malloc( ). Each application has its own address space, and thus its own heap, and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

Malloc 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 malloc using free( ) so you cannot assume there is a single used region followed by some free memory.



This can be improved upon… Adding 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, 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.