Assignment 3 - friendship ended with malloc

Due: 11:59PM Wednesday 28th April 2021 Sydney time

This assignment is worth 15% of your final assessment

Task description

In this assignment you will be implementing a simple dynamic memory allocator with a similar interface to the standard libary functions (such as malloc) that you are familiar with.

You will be implementing your allocator as a library of functions that can be called by other programs, rather than as a standalone executable.

Introduction

The standard library and other real world allocator implementations obtain raw areas of memory from the kernel using the `mmap` and `brk` syscalls. Under a simplified memory model, these raw areas of memory are described as "the heap". The allocator implements internal data structures that keep track of which parts of these raw blocks of memory are allocated or not. As you will be familiar, programs use library functions such as `malloc` and `free` to request dynamic memory from and return dynamic memory to the allocator. It is the allocator's responsibility to use its internal data structures to mark memory that is allocated so that it does not overlap with any other allocations made, and to allow memory that is `free`d to be allocated to future dynamic memory requests. Furthermore, it is the allocator's responsibility to appropriately call syscalls to obtain further memory from the operating system if required for an allocation, or to return memory to the operating system if it is no longer required.

In this assignment, your allocator will manage a virtualised heap using a function we provide that simulates using `brk` to manipulate a real process heap. When dynamic memory is requested from your allocator, it will allocate it from this virtual "heap".

Memory allocation process

Your virtual heap is simply a contiguous region of memory that you have read and write access to. Details on how you manage it are below in "Managing your Virtual Heap Memory".

You will implement a simple buddy allocation algorithm to manage your virtual heap. Buddy allocation fulfils all allocations from a starting block of memory that is 2^n bytes in size, where n is a non-negative integer. During the allocation process, the starting memory is repeatedly halved, creating blocks of size 2^i bytes where i is a non-negative integer and i < n. i also has a minimum value that we will refer to as MIN.

Allocation Algorithm

When a request for an allocation of k bytes of memory is made, follow this algorithm:

1. If there exists an unallocated block of size 2^j bytes such that $2^{j-1} < k \leq 2^j$, allocate and return the leftmost such unallocated block. Exception: if there exists an unallocated block of size 2^{MIN} and $k \leq 2^{MIN}$, allocate and return the leftmost such block (because there are no smaller blocks to allocate)
2. If there exist no such unallocated blocks, create an unallocated block of size 2^j bytes (if $k \leq 2^{MIN}$, let j = MIN) by the below steps

3. Split the leftmost unallocated block of size 2^{j+1} bytes in half. Allocate and return the left half for this request. The right half becomes a free unallocated block of size 2^j. Blocks which are split are not allocatable. The 2 child blocks which result from a split are called buddies of each other.
4. If no unallocated block of size 2^{j+1} exists, repeat the last step with the leftmost unallocated block of size 2^{j+2}, and so on as required, up to splitting the original block of 2^n bytes.
5. If no block of suitable size can be found, return an appropriate error as described below in the functions you have to implement

Note the following:

- The entire unallocated block is allocated for the request. In buddy allocation, there is often some wasted space because requested memory is less than the appropriate power-of-2 block size.
- "Left" refers to smaller memory addresses
- Any area of memory can only be allocated once. If allocation succeeds, callers will expect to have exclusive access, at the pointer you return, to the number of bytes of memory they requested.
- Size limits are defined by the types we have specified for the functions you implement.

Deallocation algorithm

When a previously allocated block of memory of size 2^j bytes is requested to be freed, follow this algorithm:

1. If the buddy block of the freed block is also unallocated, merge the two buddies to form an unallocated block of size 2^{j+1}.
2. Repeat the previous step if the buddy block of this new 2^{j+1} block is also unallocated. Continue until no more unallocated buddy blocks can be merged.

Note the following:

- Unallocated buddy blocks which have been merged are no longer eligible for allocation, only the new parent unallocated block can be allocated (unless it is split up again)

Reallocation algorithm

When an allocated piece of memory is requested to be reallocated, follow this algorithm:

1. Make a new request for allocation at the new requested size, computing as if the previous piece of memory is freed
2. If the new allocation succeeds, the original data in the previous piece of memory should be made available at the new allocated block according to the details for the `virtual_realloc` function below
3. If the new allocation fails, the original allocation must be unchanged

Managing your Virtual Heap Memory

All functions that you have to implement accept a `heapstart` parameter, which is a pointer to the start of the contiguous region of memory that represents your virtual heap.

Your code can call a `virtual_sbrk` function that you can use to determine and change the size of your virtual heap space, analogously to the real-world `sbrk` and `brk` syscalls; its prototype is below. You do not need to use the real `sbrk` or `brk` for this assignment.

```
void * virtual_sbrk(int32_t increment);
```

The "program break" of your virtual heap refers to the address of the first byte after the end of your heap. (For the avoidance of doubt, "program break" in this document always refers to your virtual heap, and not any real program break of your process memory layout).

The `increment` parameter indicates the number of bytes to increase (positive increment) or decrease the virtual heap size, by changing the program break by the same amount. If the call is successful, `virtual_sbrk` returns the previous program break of your virtual heap. If the call is unsuccessful (for example because the virtual heap cannot increase further in size), `virtual_sbrk` returns `(void *)(-1)` (`errno` is not set).

If `virtual_sbrk` indicates it cannot increase your virtual heap space and this would cause your allocation to fail, then you should return the appropriate error for your function.

Functions to implement

Implement the following functions for your allocator. Do not write any `main()` function. Other programs will directly call your functions.

```
void init_allocator(void * heapstart, uint8_t initial_size, uint8_t min_size);
```

This function will be called exactly once before any other functions in your allocator are called.

In this function, initialise your buddy allocation data structures and complete any preparation in the virtual heap memory you have been provided. This will be passed to each of your allocator functions, using the `heapstart` pointer only. You cannot pass any other state between your functions other than what is in your virtual heap. You may not use the standard library's dynamic memory (such as malloc), or global variables, or files (even if you store pointers to external memory within your virtual heap).

Your buddy allocator starts with an initial unallocated block of memory of $2^{initial\_size}$ bytes. The minimum size of allocatable blocks will be 2^{min_size}.

It is up to you how you lay out your data structures in your virtual heap, but it must semantically behave as the buddy allocator described above. Use `virtual_sbrk` to set your virtual heap size as you require.

```
void * virtual_malloc(void * heapstart, uint32_t size);
```

Request an allocation from your allocator of `size` bytes. Follow the buddy allocation algorithm outlined to return a pointer to the block of memory that you have allocated for the caller. Return `NULL` if you cannot fulfil the allocation or if `size` is 0. Newly allocated memory does not need to be initialised.

```
int virtual_free(void * heapstart, void * ptr);
```

`ptr` is a pointer to a previously allocated block of memory. Your allocator should free the allocation according to the buddy allocation algorithm above. If successful, return 0. If `ptr` is not a pointer to a block of memory that was previously allocated, return non-zero.

```
void * virtual_realloc(void * heapstart, void * ptr, uint32_t size);
```

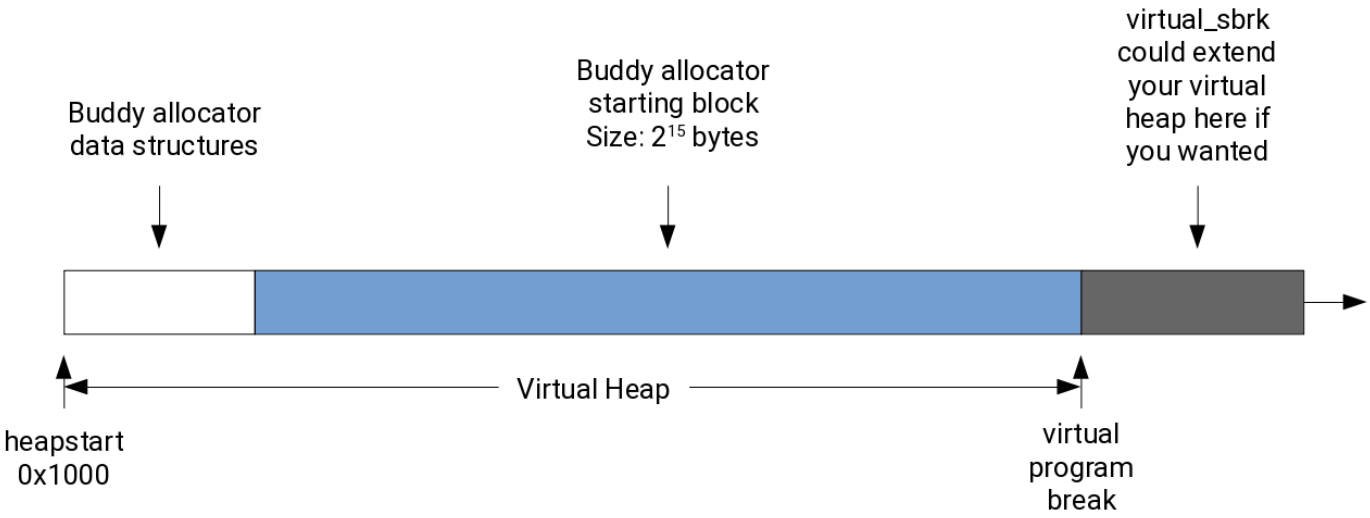
Resize a previously allocated block of memory pointed to by `ptr` to a new size of `size` bytes, according to the buddy allocation algorithm above. The contents of the new block of memory should be identical to the old. If the size is smaller, the contents should be truncated. If the size is larger, the newly added memory region does not need to be initialised. Return the pointer to the new allocation (which may be identical to `ptr`). Return `NULL` if you cannot fulfil the reallocation. In this case, the previously allocated block of memory should not be freed and should be unchanged. If `ptr` is `NULL`, you should behave as if `virtual_malloc(size)` was called. If `size` is 0 (including if `ptr` is `NULL` in this case), you should behave as if `virtual_free(ptr)` was called.

```
void virtual_info(void * heapstart);
```

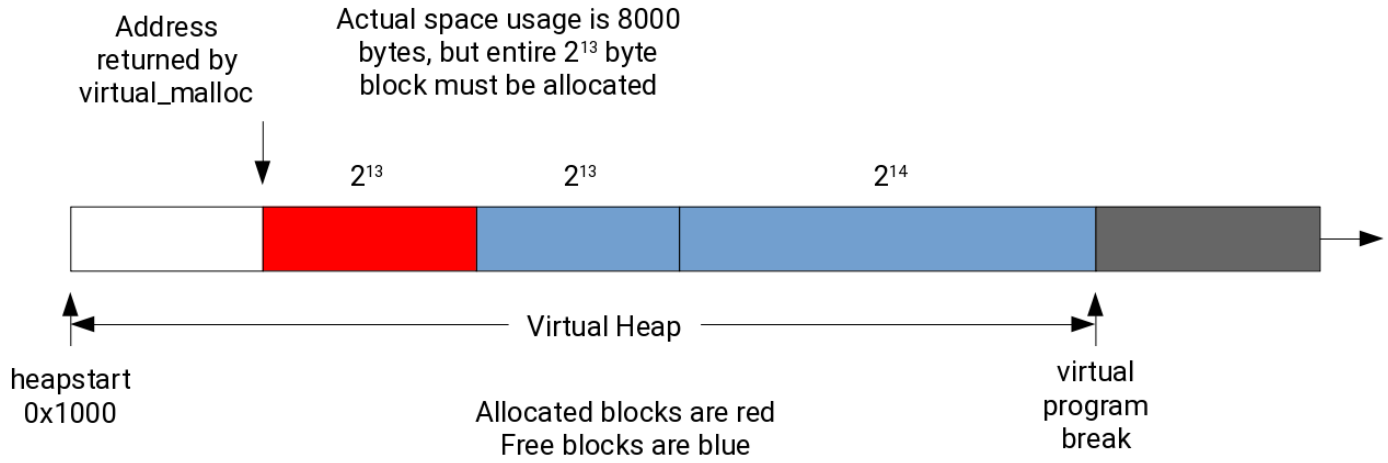
Print out information about the current state of your buddy allocator to standard output. Output one line per allocatable block, allocated or unallocated, from "left" to "right", in the following format: `<allocation status> <size>`. Allocation status is either `allocated` or `free` and size is in bytes (of the entire block).

Example

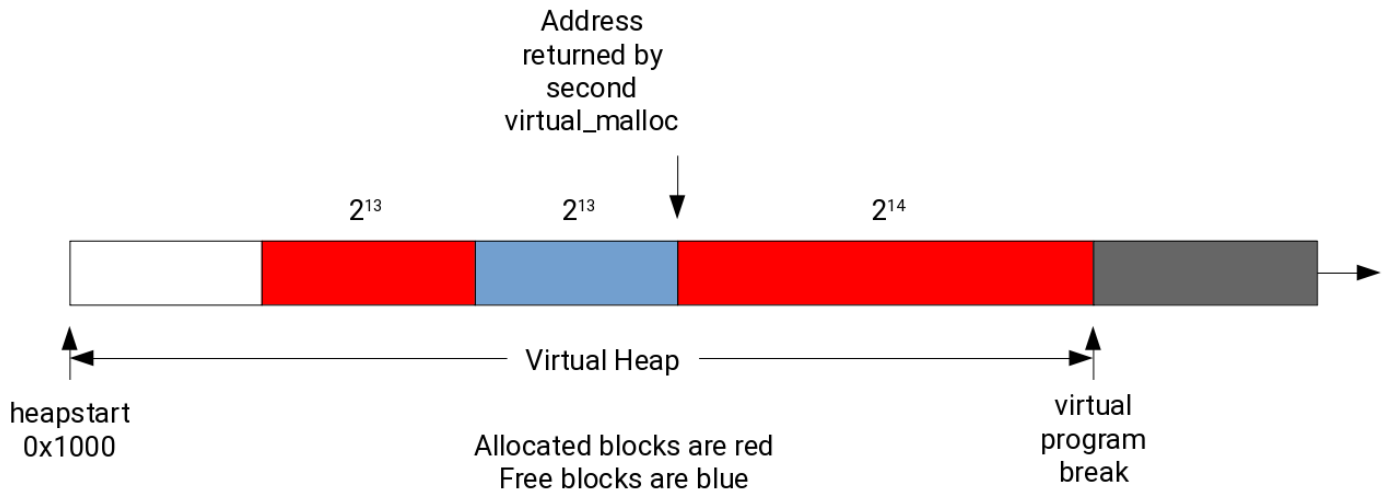
Suppose that `heapstart = 0x1000` and `init_allocator` is called with `initial_size = 15` and `min_size = 12`. The diagram below shows the initial state of the virtual heap. Note that the space and location of your data structures is up to you and depends on your implementation of the buddy allocator. Note that `virtual_sbrk` has been used to set the virtual program break appropriately to fit what is being placed on the virtual heap.



Suppose that `virtual_malloc(heapstart, 8000)` is called. $2^{12} < 8000 \leq 2^{13}$, but we only have an unallocated block of size 2^{15}. Therefore, we split the starting block in half twice, then allocate the leftmost block of size 2^{13} for this request. The address of this block is returned to the caller.



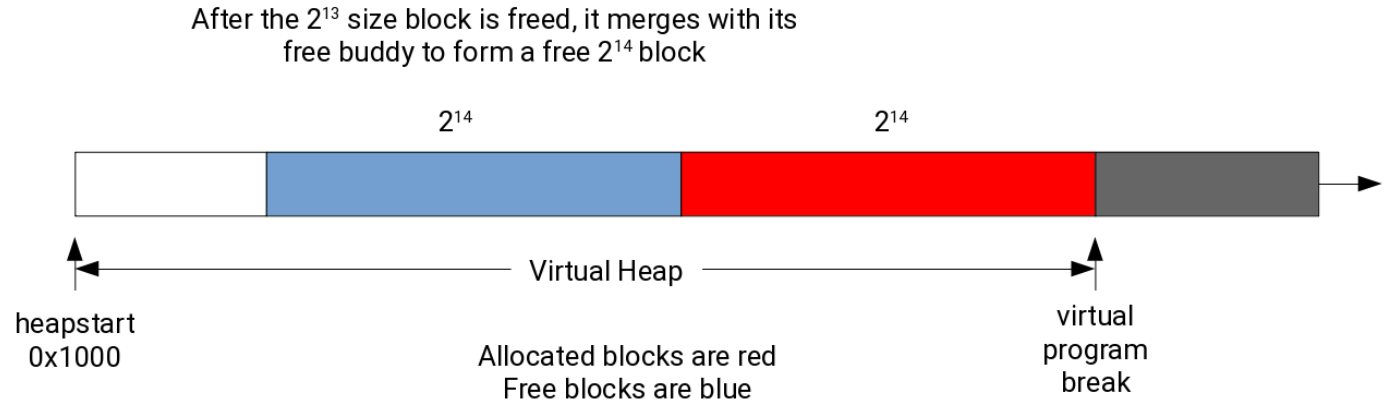
Suppose that `virtual_malloc(heapstart, 10000)` is called. $2^{13} < 10000 \leq 2^{14}$, so we allocate our only unallocated 2^{14} block and return its address to the caller.



At this stage, if `virtual_info(heapstart)` was called, we expect the output:

```
allocated 8192
free 8192
allocated 16384
```

Suppose the allocated 2^{13} block were freed. It will merge with its unallocated buddy to the right, forming an unallocated 2^{14} size block. However, the buddy of this is not free, so there is no more merging that occurs. If the allocated 2^{14} size block were also freed, it would merge with its buddy to reform the original 2^{15} size block.



Restrictions

In your allocator library code:

- You may not access outside the bounds of your virtual heap without using `virtual_sbrk` to resize it properly
- You may not use dynamic memory of any kind, including any from the standard library allocator, `brk`, `sbrk`, `mmap`, `alloca` or variable-length arrays, except that which you obtain from `virtual_sbrk`.
- You may not access any files, or use any global or static variables.

In your testing code (code that sets up and runs test cases which call your allocator library code):

- The above restrictions do not apply.
- You may use dynamic memory. You may not use `alloca` or variable-length arrays.

If your submission violates any of these restrictions, it may receive 0.

Test Cases

You must write test cases for this assignment. Details on execution and marking of your test cases is included below.

You will need to create your own virtual heap memory area and write your own `virtual_sbrk` function in your testing code for your library to access.

Code Submission and Marking Criteria

Submit your assignment on Ed via git. It must compile and run on the Ed submission system.

Your code will be compiled with the default rule of the `Makefile`; a scaffold is provided. The scaffold contains `tests.c` which includes sample `main()` code that calls into your library functions. The executable that you use for your test cases must be built by this rule.

The default rule is also what will be used to compile your code for automarked correctness, but (only) your `tests.c` will be replaced.

Test cases you write must be executed by `make run_tests`, which should run your tests and report back on their results in a human readable format.

You can modify your Makefile, but you cannot change the compiler, the compilation flags in `CFLAGS`, or remove `tests.c` from the default rule.

Failing to adhere to these rules will prevent your markers from running your code and tests and you may receive 0.

Only the last submission will be graded. Late submissions will incur the late penalty described in the Unit of Study outline.

The marking criteria follow (15 marks total):

- 4 marks for correctness (passing Ed automatic test cases). Some test cases will be hidden and not available before or after the deadline.
- 2 marks for minimising the amount of virtual heap memory you use under several allocation scenarios. You are only eligible for these marks if you pass all the correctness test cases. Usage will be measured after a series of allocations are performed, not while your allocation functions are executing. All submissions using less than defined cut-off amounts of memory will be awarded 2 marks, with the amount awarded decreasing with increasing memory usage above this level.
- 7 marks for an oral solution discussion with a COMP2017 staff member regarding your implementation. You will be required to attend a Zoom session after the submission deadline according to allocation instructions sent to you via email and posted on Ed. In this session, you may be asked:
  - Explain how you are maintaining the state of the allocation and your memory layout
  - Explain how `virtual_malloc` and/or `virtual_free` and/or `virtual_realloc` change your state and memory layout
  - Discuss the overhead and space efficiency of how you are maintaining state
  - Further questions
  - Your code will also be assessed on C coding style conventions, provided in Ed resources
  - If you do not make a reasonable attempt at the assignment, you will not be eligible for an oral session.
- 2 marks for your own automated test cases, executed with `make run_tests` as described above. Ensure that your testing code outputs the result of each test in a human readable format.

Warning: Any attempts to deceive or disrupt the marking system will result in an immediate zero for the entire assignment. Negative marks can be assigned if you do not follow the assignment problem description or if your code is unnecessarily or deliberately obfuscated.

Hints

- There are marks available for minimising virtual heap memory usage in this assignment. Consider how you implement your data structures and make sure you shrink your virtual heap with `virtual_sbrk` when you can. Remember these marks are only awarded once you pass all correctness test cases.

Academic Declaration

By submitting this assignment you declare the following:

I declare that I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure, and except where specifically acknowledged, the work contained in this assignment/project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgment from other sources, including published works, the Internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

I acknowledge that the School of Computer Science, in assessing this assignment, may reproduce it entirely, may provide a copy to another member of faculty, and/or communicate a copy of this assignment to a plagiarism checking service or in-house computer program, and that a copy of the assignment may be maintained by the service or the School of Computer Science for the purpose of future plagiarism checking.