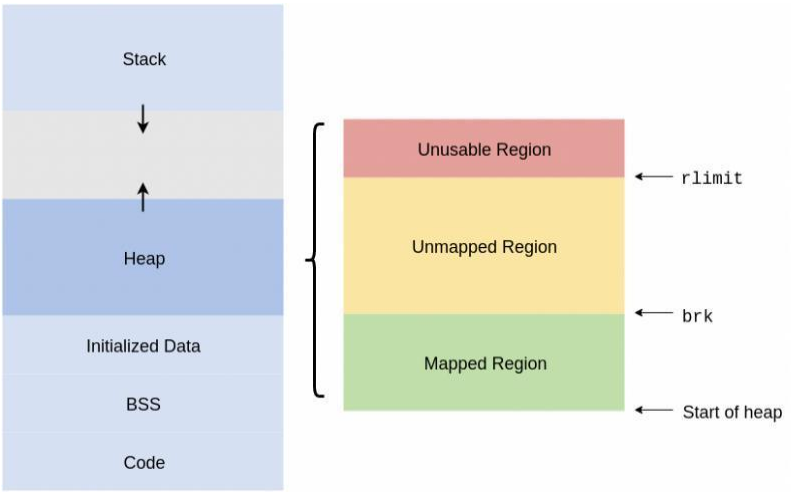
# 1. Background: Getting Memory from the OS

## 1.1 Process Memory

Each process has its own virtual address space. Parts of this address space are mapped to physical memory through address translation. In order to build a memory allocator, we need to understand how the heap in particular is structured.




The heap is a continuous (in terms of virtual addresses) space of memory with three bounds:

- The bottom of the heap.
- The top of the heap, known as the break. The break can be changed using **brk** and **sbrk**. The break marks the end of the mapped memory space. Above the break lies virtual addresses which have not been mapped to physical addresses by the OS.
- The hard limit of the heap, which the break cannot surpass (managed through **sys/resource.h's** functions **getrlimit(2)** and **setrlimit(2))**

Here we'll be allocating blocks of memory in the mapped region and moving the break appropriately whenever you need to expand the mapped region.

## 1.2 sbrk

Initially the mapped region of the heap will have a size of 0. To expand the mapped region, we have to manipulate the position of the break. The recommended syscall for doing this is **sbrk**.

```
void *sbrk(int increment);
```

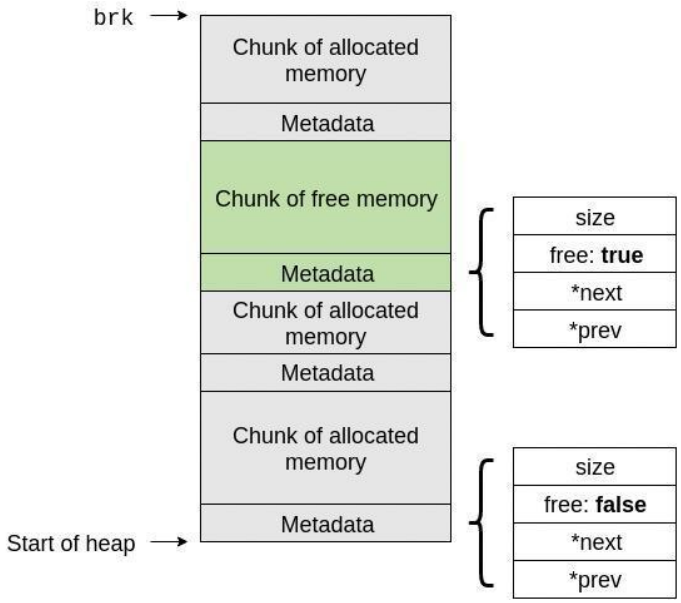
**sbrk** increments the position of the break by increment bytes and returns the address of the previous break (i.e. the beginning of newly mapped memory). To get the current position of the break, pass in an increment of 0. For more information, read the man page.

## 1.3 Heap Data Structure

A simple memory allocator can be implemented using a linked list data structure. The elements of the linked list will be the allocated blocks of memory on the heap. To structure our data, each allocated block of memory will be preceded by a header containing metadata.

For each block, we include the following metadata:

- **prev, next**: pointers to metadata describing the adjacent blocks
- **free**: a boolean describing whether or not this block is free
- **size**: the allocated size of the block of memory




You might also consider using a [zero-length array] to serve as a pointer to the memory block.

# 2 Implementation of Memory Allocator

There are many ways to structure a memory allocator. We will be implementing a memory allocator using a linked list of memory blocks, as described in the previous section. In this section, we'll describe how allocation, deallocation, and reallocation should work in this scheme.

## 2.1 Allocation

**void *mm_malloc(size_t size);**
The user will pass in the requested allocation size. Make sure the returned pointer is pointing to the beginning of the allocated space, not your metadata header.

One simple algorithm for finding available memory is called **first fit.** When your memory allocator is called to allocate some memory, it iterates through its blocks until it finds a sufficiently large free block of memory.

- If no sufficiently large free block is found, use **sbrk** to create more space on the heap.
- If the first block of memory you find is so big that it can accommodate both the newly allocated block and another block in addition, then the block is split in two; one block to hold the newly allocated block, and a residual free block. If it's a bit larger than what you need, but not big enough for a new block (i.e. it's not big enough to hold the metadata of a new block), be aware that you will have some unused space at the end of the block.
- Return **NULL** if you cannot allocate the new requested size.
- Return **NULL** if the requested size is 0.
- For ease of grading, we ask that you zero-fill your allocated memory before returning a pointer to it.

## 2.2 Deallocation

**void mm_free(void *ptr);**

When a user is done using their memory, they'll call upon your memory allocator to free their memory, passing in the pointer **ptr** that they received from **mm_alloc**. Deallocating doesn't mean you have to release the memory back to the OS; you'll just be able to allocate that block for something else now.

- As a side-effect of splitting blocks in your allocation procedure, you might run into issues of **fragmentation**: when your blocks become too small for large allocation requests, even though you have a sufficiently large section of free memory. To solve this, you must **coalesce** consecutive free blocks upon freeing a block that is adjacent to other free block(s).
- Your deallocation function should do nothing if passed a **NULL** pointer.

## 2.3 Reallocation

**void* mm_realloc(void* ptr, size_t size);**

Reallocation should resize the allocated block at ptr to size. A suggested implementation is to first free the block referenced by **ptr**, then **mm_alloc** a block of the specified size, and finally **memcpy** the old data over. The new extended area should be zero-filled. Make sure you handle the following edge cases.

- Return **NULL** if you cannot allocate the new requested size. In this case, do not modify the original block.
- **realloc(ptr, 0)** is equivalent to calling **mm_free(ptr)** and returning NULL.
- **realloc(NULL, n)** is equivalent to calling **mm_alloc(n).**
- **realloc(NULL, 0)** is equivalent to calling **mm_alloc(0),** which should just return NULL.
- Make sure you handle the case where the argument size is less than the original size.