CS 2110 Lab 24
Intro to Malloc

The TAs :)

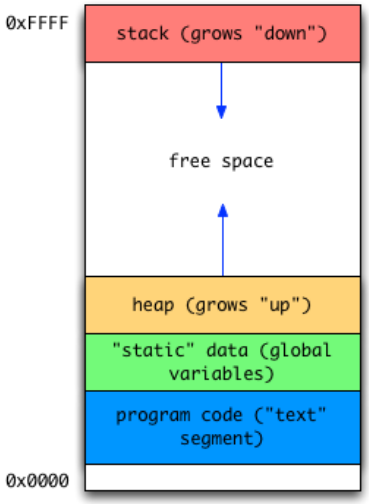
11/19/2018

1 Recitation Outline

1. What is Malloc
2. The Freelist
3. The Block
4. The Canary
5. Making Calls To Malloc

2 What is Malloc

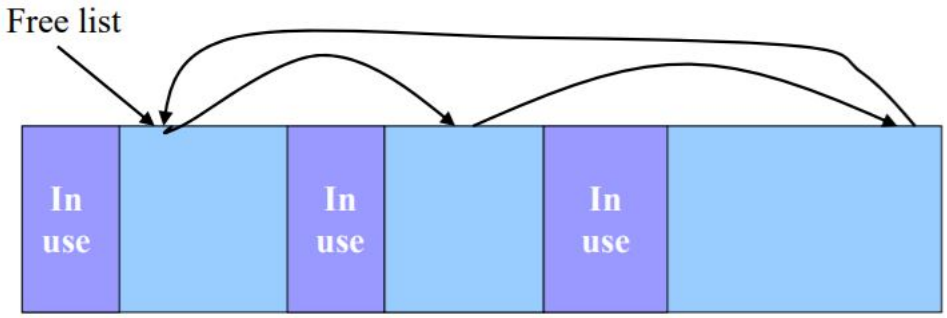
From earlier recitations, we have presented the following view of memory:



Our memory manager controls all of the memory located inside of the heap. Whenever malloc() is called, the memory manager allocates a portion of the heap to return to the user. similarly, with a call to free, the memory manager will re-obtain the portion of memory previously allocated to the user.
As seen in the image above, the heap and the stack share a common address space, with the heap growing up and the stack growing down. The stack grows down by adding local variables, while the heap grows up by allocating more memory to the program. This memory is allocated via the system call sbrk(), and the 'brk' marks the top of the heap.

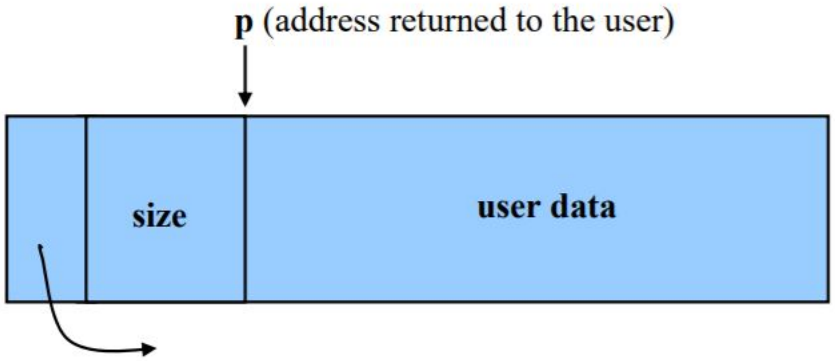
3 The Freelist

How does the memory manager control the portion of memory it is in charge of? Specifically, after many calls to malloc() and free(), our heap will not be one continuos memory block controlled by the memory manager. To organize this, the memory manager maintains a list of free blocks that are ready to be allocated to the user. This list is called the freelist and is typically implemented as a linked list ordered by either address or size (or address and size, but that will be covered next week).



4 The Block

Each individual block in the freelist has information about it, including its position in the freelist and its size. This is known as the metadata, and is stored within the block of memory. More formally, it holds information such as a pointer to the next block in the freelist, size of the block, and space for user data.



In this graphic, the first arrow is a pointer pointing to the next block in the free list, the size is where the size of the block is stored, and **p** is the address that we will want to return to the user when they call malloc since the user does not care about the metadata. (If block b is of type metadata_t*, we can easily find the address of p by doing b + 1 to skip over the metadata).

5 The Canary

When a user has a block of code, there isn't anything stopping them from overwriting their block's bounds. When this occurs though, another block could be overwritten and general complications will arise. In order to prevent a user from writing more than what they control, c uses canaries for its buffer overflow protection technique.

In c, we can monitor for data corruption by checking to see if blocks of data allocated to the user have been overflowed by verifying canary values. Canaries are unique to the block of data they buffer and wrap around the head and tail ends of the data. If a canary value has been modified from what it is supposed to be (this can be done by checking to see if the head canary == the tail canary), then memory from elsewhere has

leaked into that block or data in the block has overflowed, corrupting it. By verifying corrupted canaries, execution of the affected program can be terminated, preventing it from misbehaving or from allowing an attacker to take control over it.

6 Making Calls to Malloc

So now that we understand the freelist, how do we actually give users blocks of memory? The freelist is essentially a linked list of memory chunks, with a pointer within the metadata pointing to the next free block in memory. When the user asks for a block of a certain size, malloc will look at the free list and find the most suitable sized block to return to the user. There are a few cases to consider:

1. The user asks for a block of size `block_size` and there is a block in the freelist of size `block_size`. This is the perfect case! We can just return the block of perfect size to the user and remove it from the freelist, and we are done.

2. The user asks for a block of size `block_size` but there isn't a perfect block in the freelist. There is, however, a block that is slightly bigger in size than `block_size`. This is fine because then we can split the block into two, return the half that is of `block_size` and leave the other half in the free list.

3. The user asks for a block of size `block_size` but there isn't a block left in the freelist that can fit such a block! There is a still a way though, and this is calling `sbrk()`. This function will return a large chunk of memory back, which can be inserted into the freelist.

Why don't we just keep calling `sbrk()` whenever we need new memory instead of having to deal with a freelist? Calling `sbrk()` requires a certain amount of overhead that is quite computationally expensive and harder to keep track of. `sbrk()` should only be called when it's absolutely necessary!