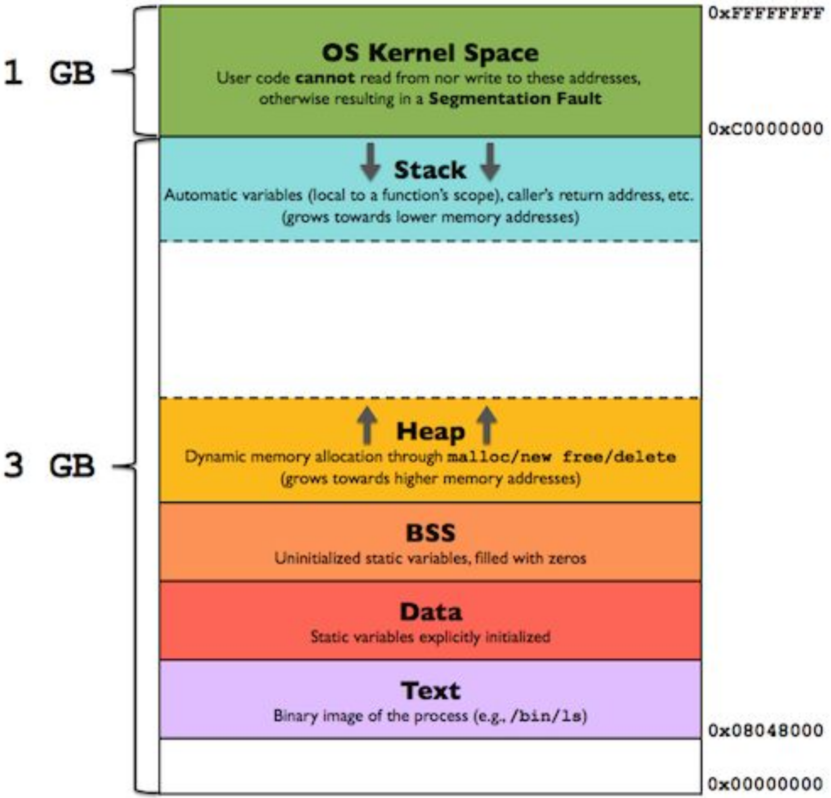
CS 24000 L04
Week 9

Malloc, Dynamic Memory, and Scope

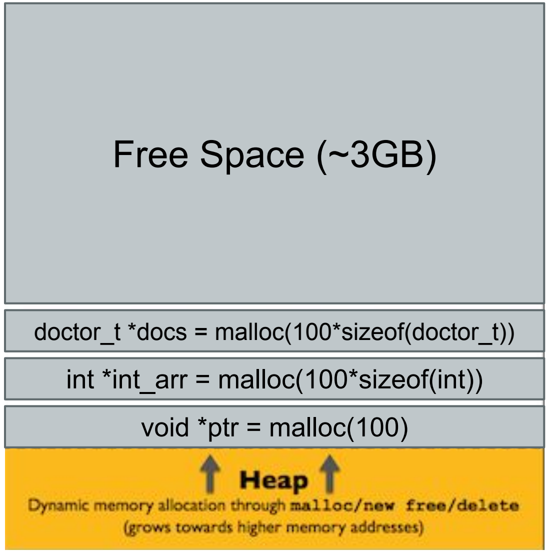
Memory Layout Review


0xFFFFFFFF
1 GB
OS Kernel Space
User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**
0xC0000000
Stack
Automatic variables (local to a function's scope), caller's return address, etc.
(grows towards lower memory addresses)
3 GB
Heap
Dynamic memory allocation through malloc/new free/delete
(grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0x08048000
0x00000000

- Local variables appear on the Stack
- Dynamic variables* appear on the Heap
- Global, uninitialized variables go in BSS
- Global, initialized variables (like format strings) go in Data
- Executable code goes in Text

* meaning those declared with malloc/free

How Malloc Works



Free Space (~3GB)

doctor_t *docs = malloc(100*sizeof(doctor_t))

int *int_arr = malloc(100*sizeof(int))

void *ptr = malloc(100)

Heap

Dynamic memory allocation through malloc/new free/delete

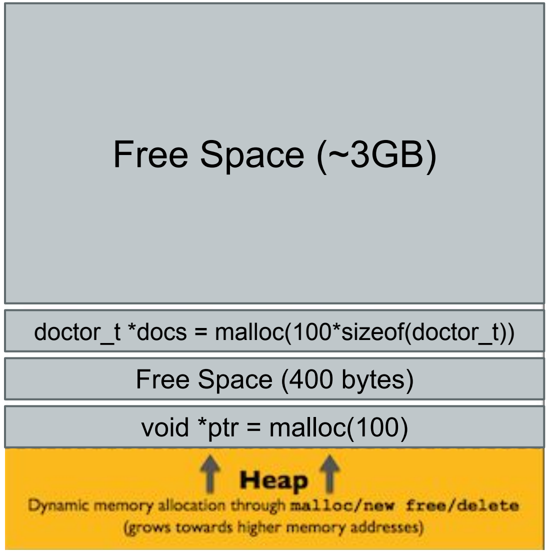
(grows towards higher memory addresses)

When you initialize a pointer using ***malloc***, the OS finds an ***unused space in the heap***, claims it, and ***returns the address*** to you.

If you never call ***free***, this memory never gets reclaimed by the OS, even if it goes out of scope. This is called a ***dangling pointer***.

Memory Fragmentation

free(int_arr) yields the following:



Free Space (~3GB)

doctor_t *docs = malloc(100*sizeof(doctor_t))

Free Space (400 bytes)

void *ptr = malloc(100)

Heap
Dynamic memory allocation through malloc/new free/delete
(grows towards higher memory addresses)

Calling free can fragment your memory space. This is expected.

Scope: Stack vs. Heap

- Any local variables are declared on the stack. These only exist in the current "stack frame" (i.e., within curly brackets{})
- Any dynamic memory allocations are on the heap. These never go out of scope, but can be lost.

Common error I’ve seen in HW8:

```
struct *ptr = malloc(sizeof(struct));

struct tmp = *ptr;

ptr2->next = &tmp;

// This creates a local copy of whatever was in ptr, not the original memory location

// Once the current stack frame ends, this copy goes out of scope and is destroyed
```