CS 521: Systems Programming

Dynamic Memory Allocation

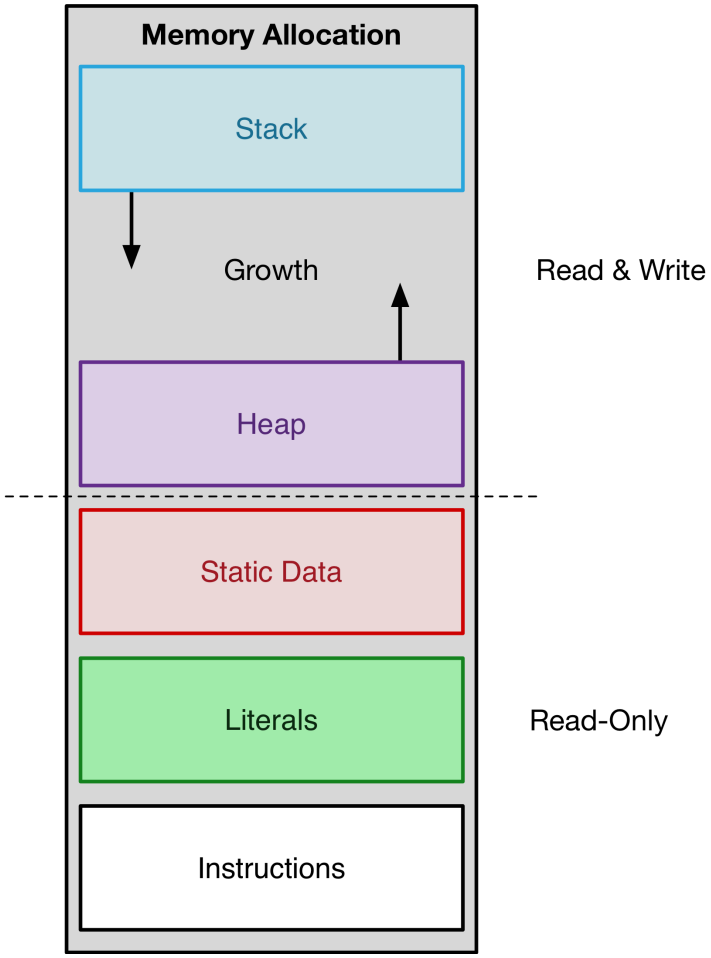
Lecture 11

Memory Allocation

- A running instance of a program is called a **process**
- Processes are allocated memory to store instructions, string literals, constants, and more
- At run time, there are two places memory is allocated:
  - Stack
  - Heap

Memory Layout

- **Stack**: Temporary data
  - Made up of stack frames
- **Heap**: long-lived data



The Stack

- Thus far, we've allocated everything to the stack
  - `int a = 5;`
- A good fit if we already know what data we're working with ahead of time
- If we know a user wants to enter, say, a number, we set aside some memory for them to do it
- If we don't know what data will be coming in ahead of time, then we need to place it on the **heap**

Demo: Returning Pointers on the Stack

- What happens if we have a function that returns a pointer to something that was stored on the stack?
  - ...

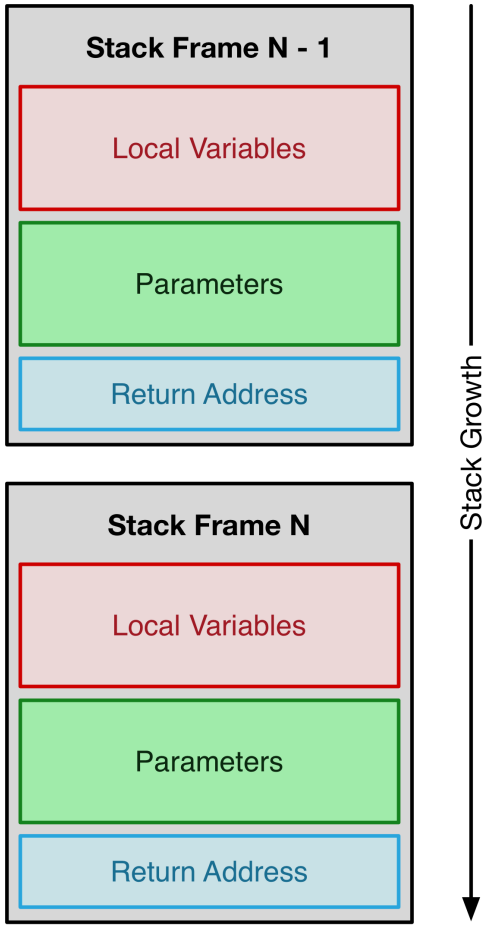
Stack Frames [1/2]

- Each function call has a stack frame
  - You may also see these called activation records
- The stack frame contains the local variables, return address, and parameters
  - In other words, the "execution environment" for each function call
- Stack frames get pushed onto the stack with each function call
  - Unchecked recursive functions can lead to stack overflow

Stack Frames [2/2]

```c
int main(int argc, char *argv[]) {
    hello(1);
    return 0;
}

int hello(int i) {
    int j = i + 1;
    printf("Hello world!\n");
    return j;
}
```



Stack Overflow

We can cause a stack overflow by making the stack grow too large.

Consider a recursive function:

```
int foo()
{
   return foo();
}
```

Heap [1/2]

- The heap is where we **dynamically** allocate memory
- This is achieved using the `malloc()` function
- Allocating memory dynamically lets us cope with changing inputs
  - Perhaps a user wants to load a file: we can't just allocate a huge variable ahead of time and hope it fits
- How would we store a file in memory anyway? There's not exactly a "file" primitive type…

Heap [2/2]

- Use dynamic memory when:
  - You need a large block of memory
  - You want to keep a variable around for a long time
- Data that has been allocated via `malloc` is basically global: if you know where it is in memory (with a pointer), then you can manipulate it from anywhere
  - …to be fair, that's true with all pointers!

Allocating Memory: malloc

```
#include <stdlib.h>
void *malloc(size_t size);
```

- This sets aside a block of memory for us to use
  - We just need to give it the size
- The memory address of the new block is returned as a pointer to anything (`void *`)
- Reminder: there is no guarantee the memory set aside is zeroed out

Freeing Memory: free

```c
#include <stdlib.h>
void free(void *ptr);
```

- Every `malloc()` must also have a corresponding `free()`
- Without freeing the memory, you introduce **memory leaks**
  - Imagine doing this inside an infinite loop
    - Or, maybe we don't have to imagine it…

Use After Free

```c
/* What happens here? */
int *i = malloc(sizeof(int));
*i = 3;
printf("%d\n", *i);
free(i);
printf("%d\n", *i);
```

Allocate and Clear: calloc

This gives us a nice, zeroed-out memory block:

```
void *calloc(size_t nmemb, size_t size);
```

Note that `calloc` assumes you want to allocate more than one member; you can always pass in `nmemb=1`, though.

Resizing an Allocation

You can request an existing block of memory to be resized:

```
void *realloc(void *ptr, size_t size);
```

WARNING: you *must* check the return address of `realloc`, because it can relocate the memory block!

```
some_ptr = realloc(some_ptr, size);
```

Valgrind

- As you start working with dynamic memory allocation, don't forget to watch out for memory leaks
- And invalid accesses
- Luckily, just like `gdb` can help us debug, `valgrind` helps us track down memory issues

Exercises

- Let's:
  - dynamically allocate an `int`, `double`, and `char`
  - dynamically allocate an array
    - print its contents before initializing it
  - resize the array
  - free everything