C, Memory, malloc, free

CS 2130: Computer Systems and Organization 1

April 10, 2023

Announcements

- Homework 8 available, due next Monday at 11pm
    - Gradescope submission available Wednesday
    - Limited number of submissions, test your code before submitting
- Lab tomorrow: Memory errors

C Reference Guide

Calling Functions

The C code

```
long a = f(23, "yes", 34uL);
```

compiles to

```
movl $23, %edi
leaq label_of_yes_string, %rsi
movq $34, %rdx
callq f
# %rax is "long a" here
```

without respect to how f was defined. It is the calling convention, not the type declaration of f, that controls this.

Calling Functions

But, if the C code has access to the type declaration of f, then it might perform some implicit casting first; for example, if we declared

```
long f(double a, const char *b, double c);
```

```
long a = f(23, "yes", 34uL);
```

then the call would be interpreted by C as having implicit casts in it:

```
long a = f((double)23, "yes", (double)34uL);
```

Calling Functions

and the arguments would be passed in floating-point registers, like so:

```
movl $23, %eax
cvtsi2sd %eax, %xmm0             # first floating-point argument

leaq label_of_yes_string, %rdi # first integer/pointer argument

movl $34, %eax
cvtsi2sd %eax, %xmm1             # second floating-point argument

callq f
# %rax is "long a" here
```

Function Declaration

```
int f(int x);
```

- Declaration of the function
- Function header
- Function signature
- Function prototype

We want this in every file that invokes `f()`

Function Definition

```
int f(int x) {
    return 2130 * x;
}
```

- Definition of the function

We only want this in **one** `.c` file

- Do not want 2 definitions
- Which one should the linker choose?

Header Files

C header files: `.h` files

- Written in C, so look like C
- Only put header information in them
    - Function headers
    - Macros
    - `typedef`s
    - `struct` definitions
- Essentially: information for the **type checker** that does not produce any actual binary
- `#include` the header files in our `.c` files

Big Picture

Header files

- Things that tell the type checker how to work
- Do not generate any actual binary

C files

- Function definitions and implementation
- Include the header files

Including Headers

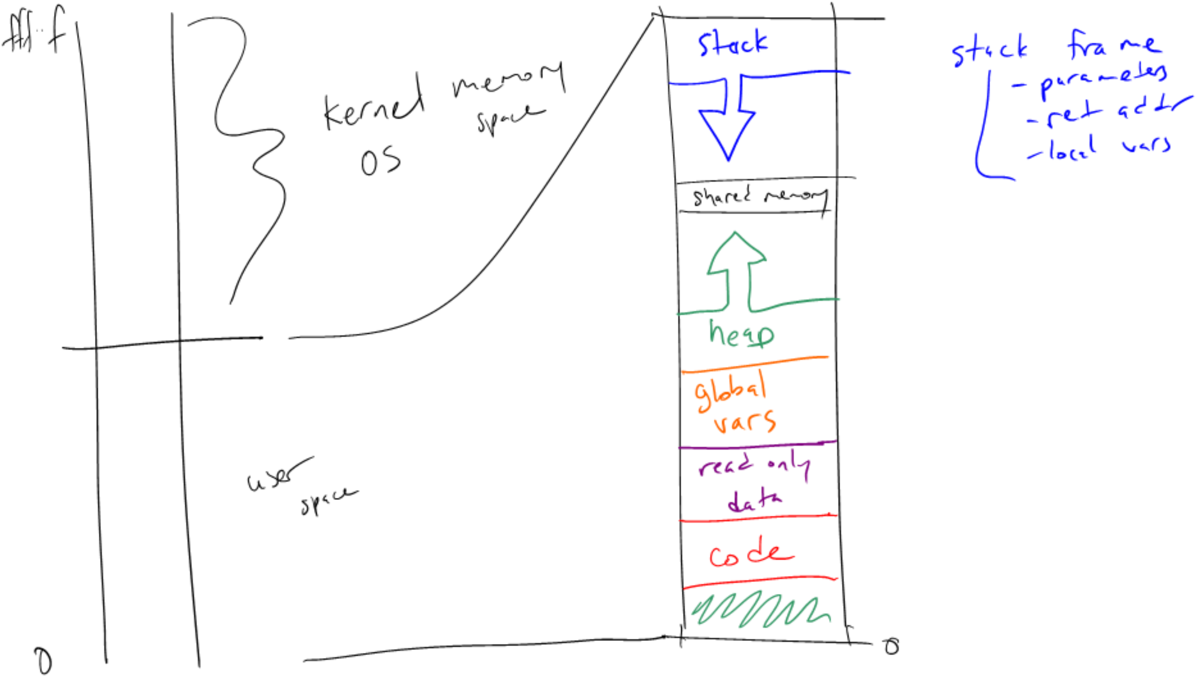
`#include "myfile.h"`

- Quotes: look for a file where I'm writing code
- Our header files

`#include <string.h>`

- Angle brackets: look in the standard place for includes
- Code that came with the compiler
- Likely in `/usr/include`

Memory



ff..f
kernel memory space
OS
user space
0
stack
shared memory
heap
global vars
read only data
code
0
stack frame
- parameters
- ret addr
- local vars

The Heap

The heap: unorganized memory for our data

- Most code we write will use the heap
- *Not a heap data structure...*

The Heap: Requesting Memory

sizeof( )

```
void *malloc(size_t size);
```

- Ask for **`size`** bytes of memory
- Returns a **`(void *)`** pointer to the first byte
- It does not know what we will use the space for!
- Does not erase (or zero) the memory it returns

malloc Example

```
typedef struct student_s {
    const char *name;
    int credits;
} student;

student *enroll(const char *name, int transfer_credits) {
    student *ans = (student *)malloc(sizeof(student));
    ans->name = name;
    ans->credits = transfer_credits;
    return ans;
}
```

The Heap: Freeing Memory

Freeing memory: `free`

```
void free(void *ptr);
```

- Accepts a pointer returned by `malloc`
- Marks that memory as no longer in use, available to use later
- You should `free()` memory to avoid *memory leaks*

An Interesting Stack Example

```
int *makeArray() {
    int answer[5];
    return answer;
}

void setTo(int *array, int length, int value) {
    for(int i=0; i<length; i+=1)
        array[i] = value;
}

int main(int argc, const char *argv[]) {
    int *a1 = makeArray();
    setTo(a1, 5, -2);
    return 0;
}
```