C Introduction

CS 2130: Computer Systems and Organization 1

Xinyao Yi Ph.D.
Assistant Professor

UNIVERSITY of VIRGINIA | ENGINEERING

Announcements

- Homework 8 due next Monday on Gradescope

Macros

```
#define NAME something else
```

- Object-like macro
- Replaces `NAME` in source with `something else`

It does this at the text level.

```
#define NAME(a,b) something b and a
```

whatever is after the comma until the second parentheses is the second thing.

Whatever is the first thing after the opening parentheses until the comma.

- Function-like macro
- Replaces `NAME(X,Y)` with `something Y and X`

text level

Lexical replacement, *not* semantic

Check the example on portal: include-me.h and use-include.c

run cpp use-include.c | less

Interesting Example

```
#define TIMES2(x)  x * 2          /* bad practice */
#define TIMES2b(x) ((x) * 2)      /* good practice */

int x = ! TIMES2(2 + 3);
```

x = ! 2 + 3 * 2; ⇒ x = 0 + 6 ⇒ x = 6.
(0) (6)

```
int y = ! TIMES2b(2 + 3);
```

x = ! ((2 + 3) * 2);

Be very explicit when you're using the macro define. Wrap them in parentheses and enforce an order of operations.

Examples and More

- header example
- `string.h`
- variadic functions

#ifndef } If ... is already defined, it won't do
#endif } anything between the ifndef and endif

Why? We want to include a header file in all of our C files. If I include the same file multiple times, it's correct but wired.

I have multiple programs running on my computer all the time. But they're all going to think they can see all of memory.

In reality, that's not true — each process only has its own isolated view of memory.

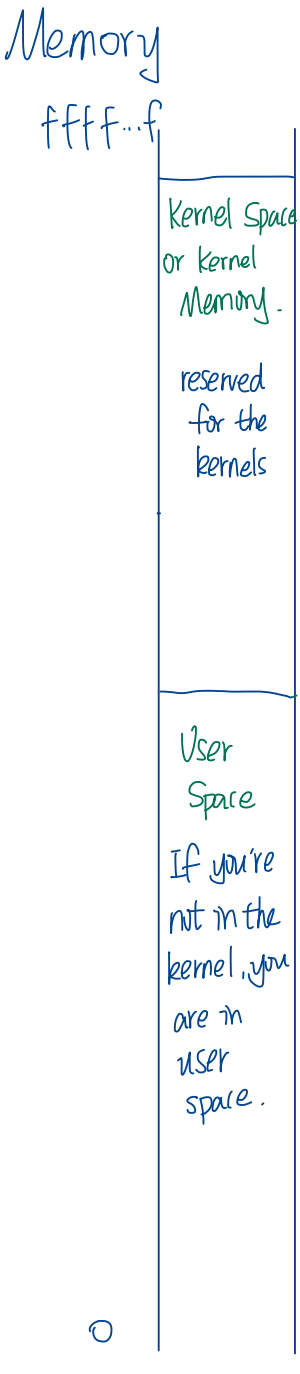
(process A and process B might both think they have memory at address 0x1000, but they're actually referring to different physical locations).

If each process thinks it has access to "all addresses" — like terabytes or exabytes of possible memory —
how is that possible when your computer might only have 16 GB of RAM?

That's the "weird" part that virtual memory solves.

We're not going to go deep into memory management right now — you'll learn about address translation, page tables, and virtual-to-physical mapping later in Computer Systems II (CSO2).

Virtual memory lets a program use **virtual addresses**,
which the hardware (Memory Management Unit, MMU) automatically **translates** into **physical addresses** —
the real locations in RAM or on disk.



From assembly:

① Generate a memory address and go to memory.

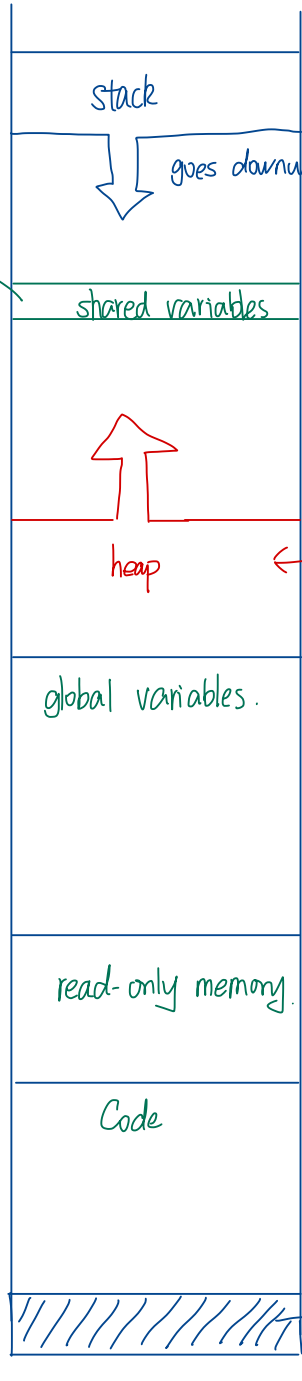
② hardware is going to check if we're actually allowed to read/write from that place in memory. (rely on a piece of software).

2 names for that: kernel or OS (Operating system).

↳ help the hardware decide whether these addresses are good or not.

It divides memory up into a couple of different segments. a chunk of memory

It's going to have different properties based on what segment of memory I'm in.



stack frame:

Anytime I call a function, there are a couple of things that get put on the stack: ① parameters
② return address.
③ local variables.

When I return, RSP moves away, memory does not get erased.

One thing missing: pointers, variables that we want to leave when our functions return. (pass to another function?)

In C, I can define variables outside of functions
For local variables in functions: put values in stack.
But for global variables, we put them here. (The size of the variables are set at compile time, and the size cannot be changed.)

Example: string literals
Things are not code, but that were in my code that are really helpful for my program.

My code is going to be stuck in memory somewhere near the bottom.

We put the code near the bottom, but not at the bottom. (Explanation next page).

At the very bottom of memory — near address 0 — there's a special protected region.
The operating system doesn't let us read or write anything there.
Why? To protect us from ourselves.

If we accidentally take an integer, cast it to a pointer, and try to access that address — for example, address 5 — the hardware will stop us.
That's why we get a segmentation fault: we tried to read or write from a memory segment we're not allowed to touch.

In C, the constant NULL is not a keyword but is usually defined as a pointer to address 0.
This means that dereferencing a null pointer — trying to access the memory at address 0 — will also trigger a segmentation fault.
And that's intentional.
We want the system to complain immediately instead of silently reading some random value from the bottom of memory.

So, the "bottom of memory" is deliberately left empty to catch common pointer mistakes —
for example, dereferencing NULL or casting an integer like 5 to a pointer and using it as an address.
When that happens, the operating system stops the program with a segmentation fault rather than letting it continue with undefined behavior.

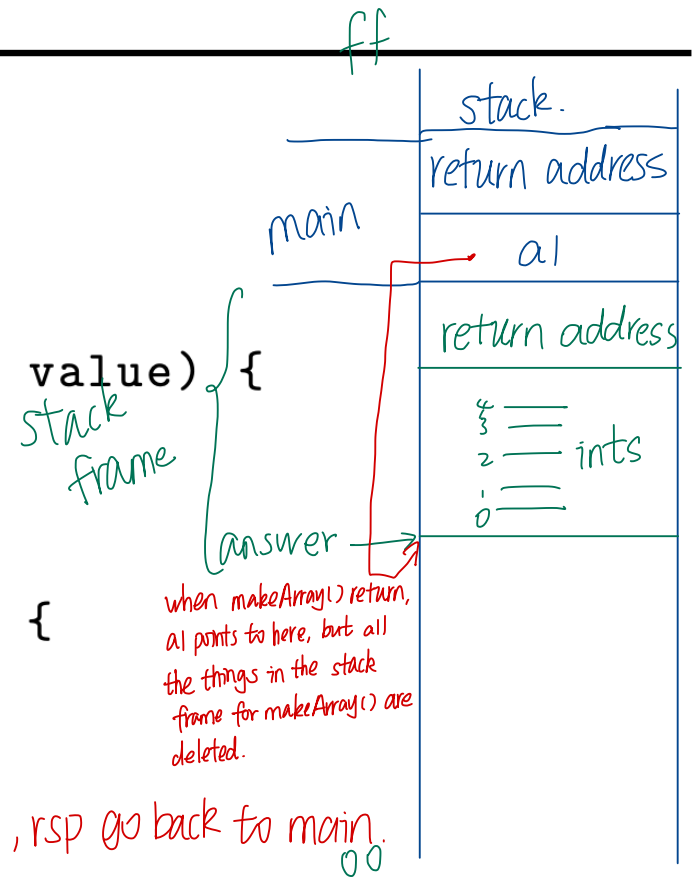
In short:

The low-address region exists as a safety barrier — it's the system's way of saying,
"You're trying to access memory that doesn't belong to you."

Memory

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;
}
```



once makeArray returns, remove everything, rsp go back to main.

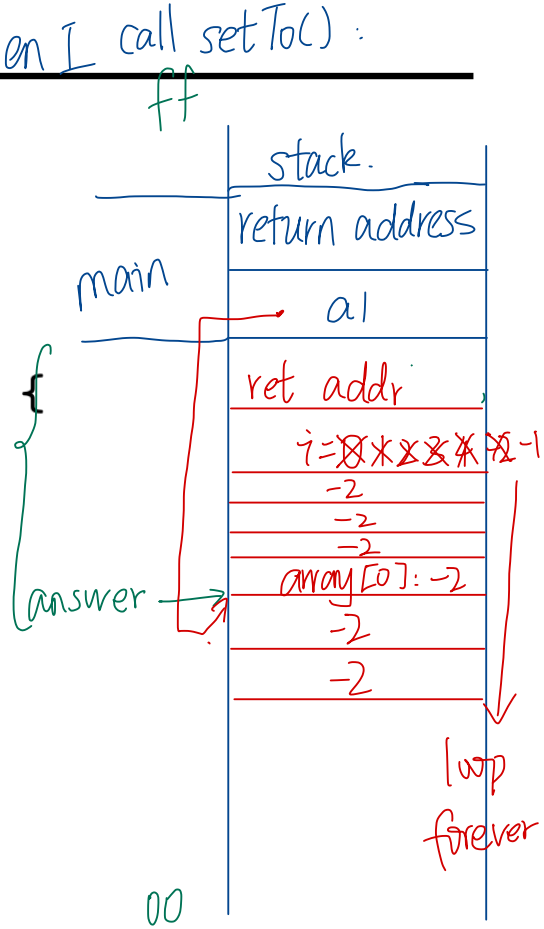
An Interesting Stack Example

```
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;
}
```

escaping pointer

in registers

stack frame



if I compile it using clang xxx.c, then it will loop forever.
But if I compile it with "-O1". same warning, but it optimize the code
maybe put "i" in register, maybe do unrolling?