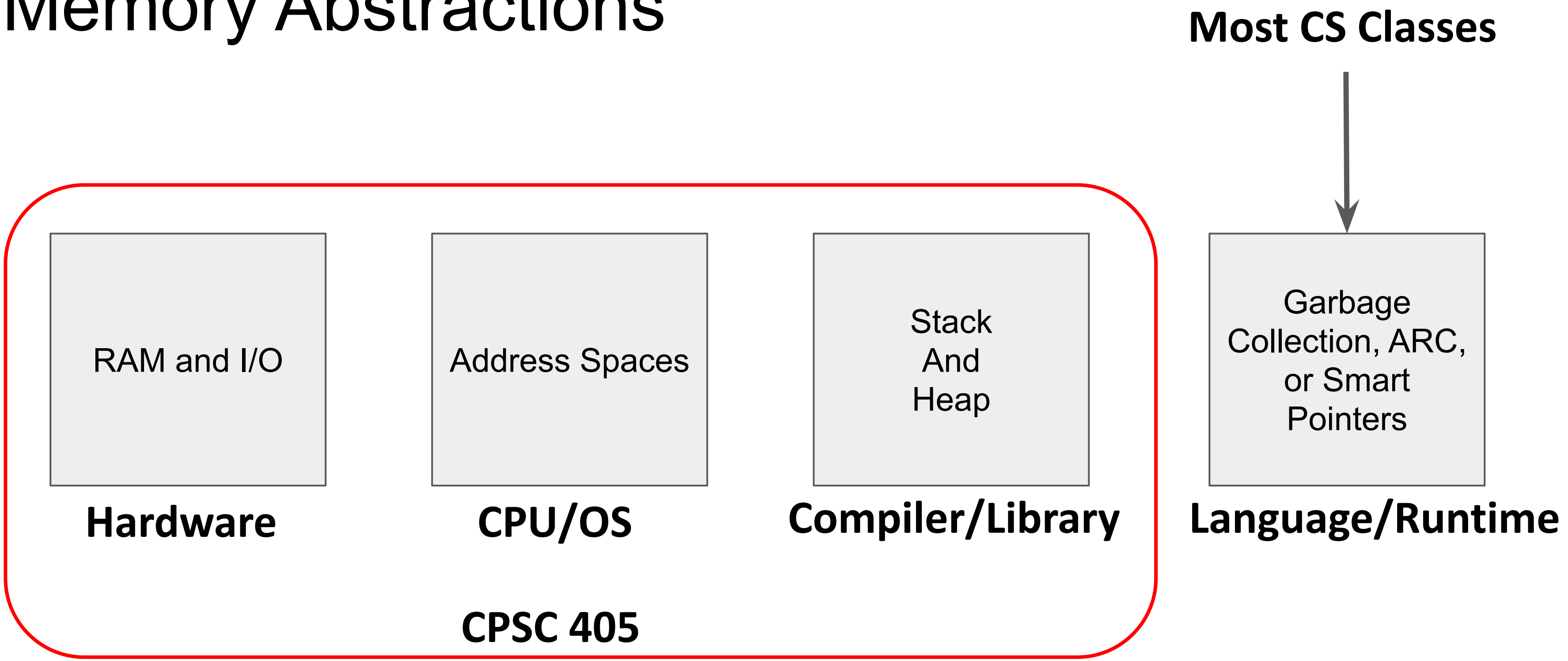
Programming Xv6 in C

Modified Charts from MIT’s 6.181

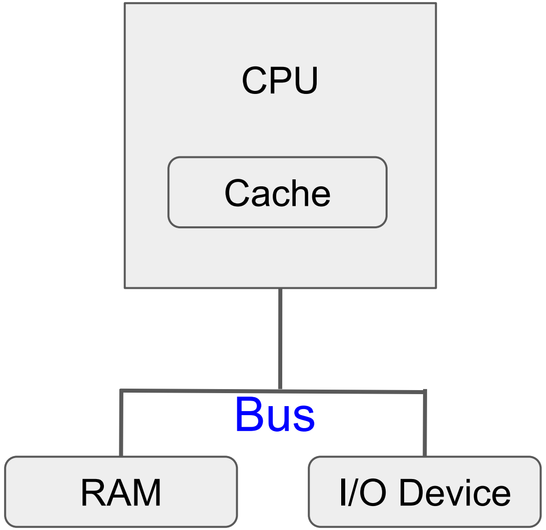
Outline

- What is memory?
- C programming basics
- Logistics
    - Don’t forget to post lecture questions before each lecture (starting this Wed.). • If you do so the night before, we’ll try to cover it in lecture
    - The first lab is due this Thursday

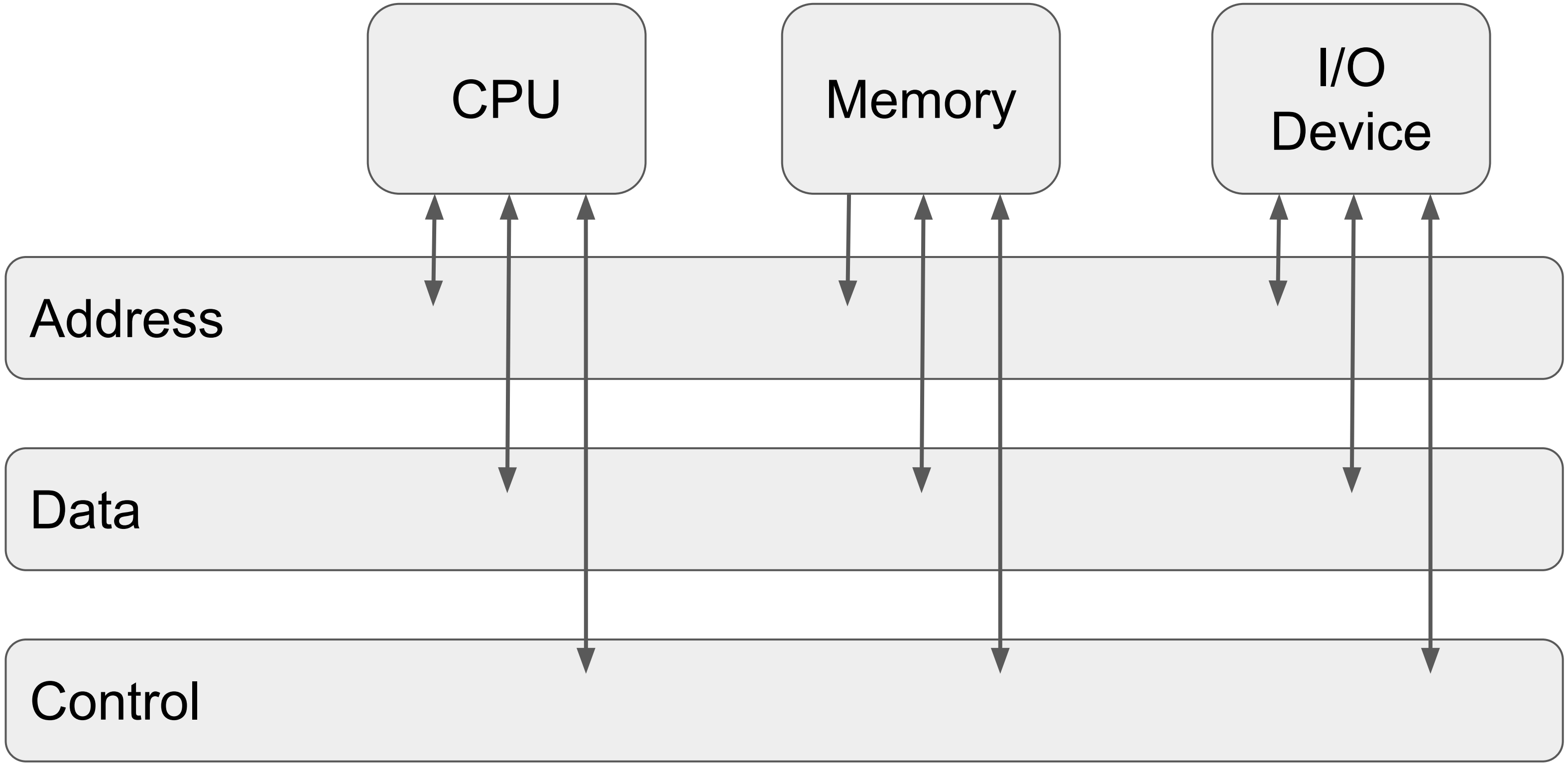
Memory Abstractions



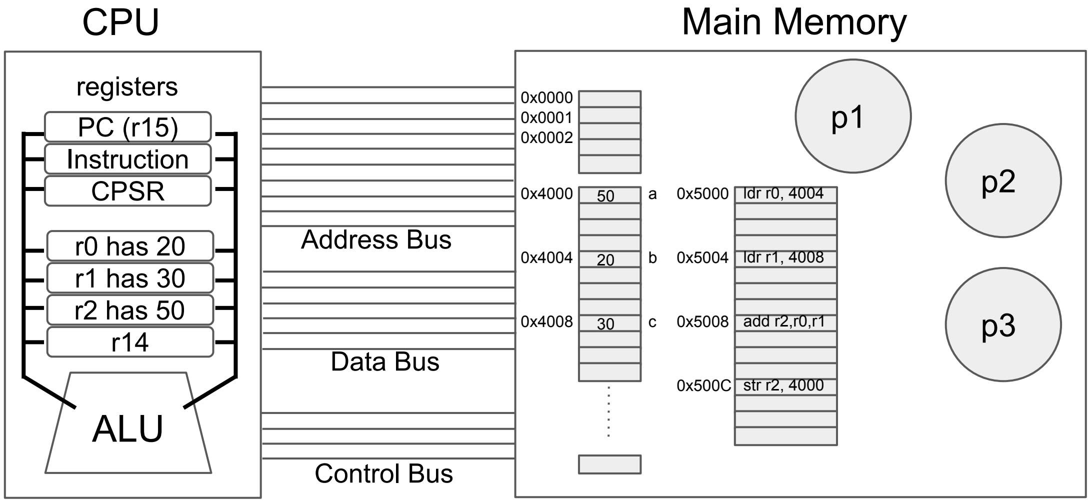
Hardware Layer

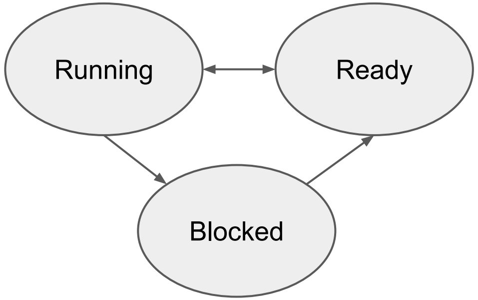


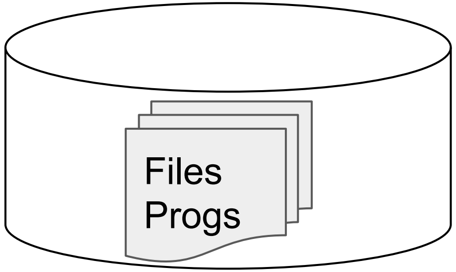
- A bus transfers data between components in the computer
- A cache remembers data previously fetched from the bus
- Speeds up the CPU by reducing the number of bus accesses
- Q: What is an IO Device?


CPU
Memory
I/O
Device
Address
Data
Control

Hardware Layer - Bus







OS shares CPU among processes

<u>C Code</u>

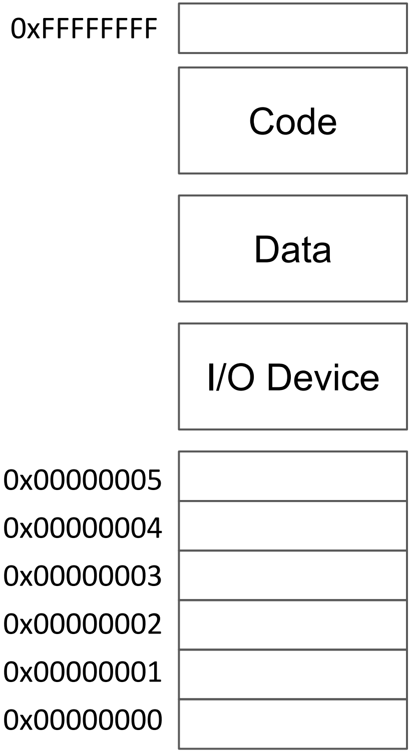
```
int a, b = 20, c = 30;
a = b + c;
```

<u>Assembly Code</u>

```
ldr r0, #0x4004
ldr r1, #0x4008
add r2, r0, r1
str r3, #0x4000
```

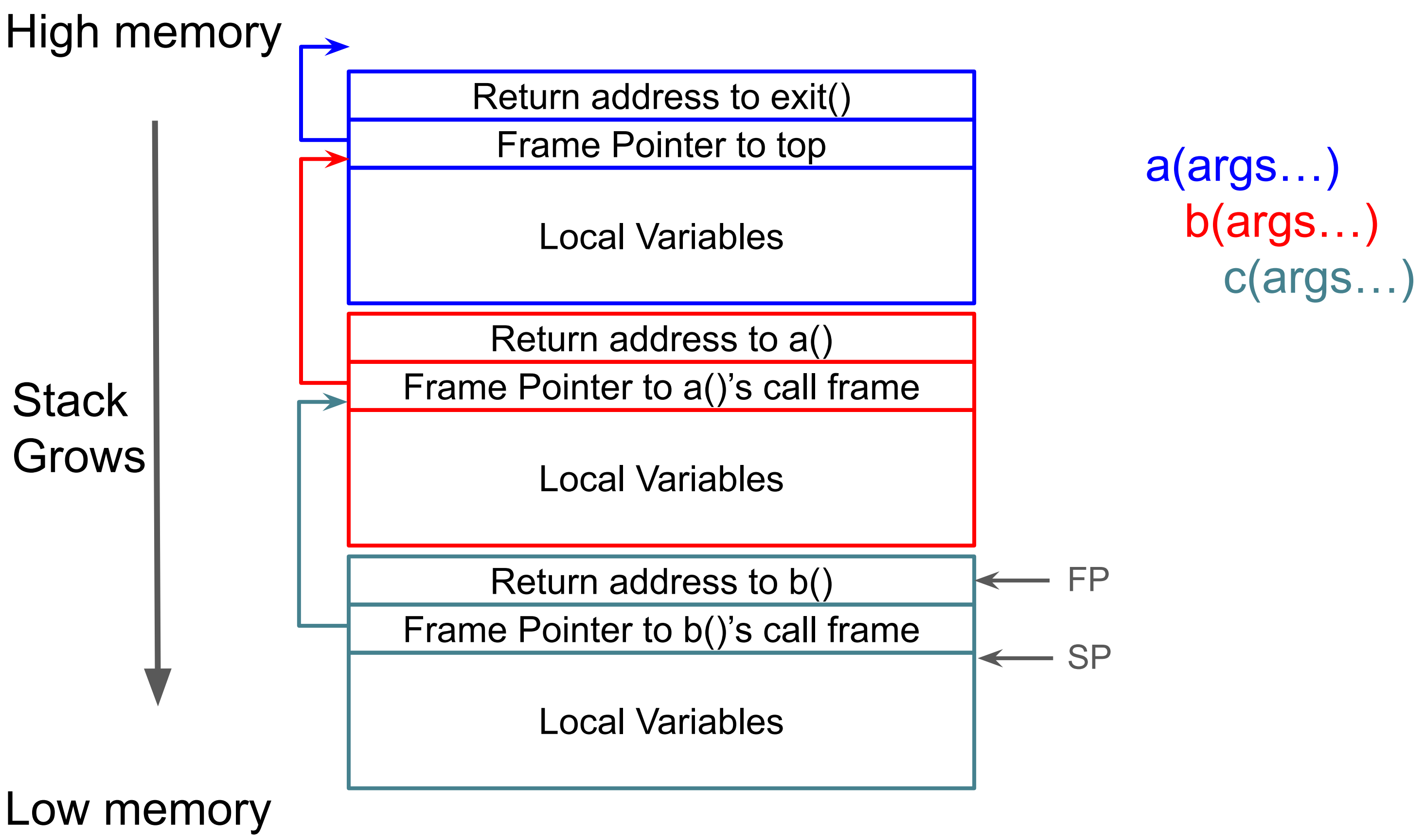
OS allocates memory for processes

Address Space - Array (or sequence) of bytes



- Array index is address of a byte (8-bits)
- Loads and stores access address space
- Address space - virtual addresses, RAM - physical addresses
- Address space is usually much larger than RAM
- Addresses that can be accessed are referred to as “mapped”
- And holes that can’t be accessed are “unmapped”
- Address space has permissions: Read - load, Write - store, Execute - allow code to execute
- Address space combines RAM and I/O devices
- What happens if the CPU loads or stores to an unmapped region?
- Why have permissions?
- What happens if the CPU loads/stores an address without permission?

Stack Basics



Heap Basics

- void *malloc(size_t size)
  - Allocates an object of size bytes
  - Returns 0 if out of memory! Otherwise, a pointer to the object.
- void free(void *item);
  - Frees an object
  - Can't be called more than once on same object
- Using heap API

```
struct foo *f = malloc(sizeof(*f));
if (!f)
    // handle out of memory error
memset(f, 0, sizeof(*f)); // initialization
// do something with f
free(f);
```

Stack and Heap in an Address Space

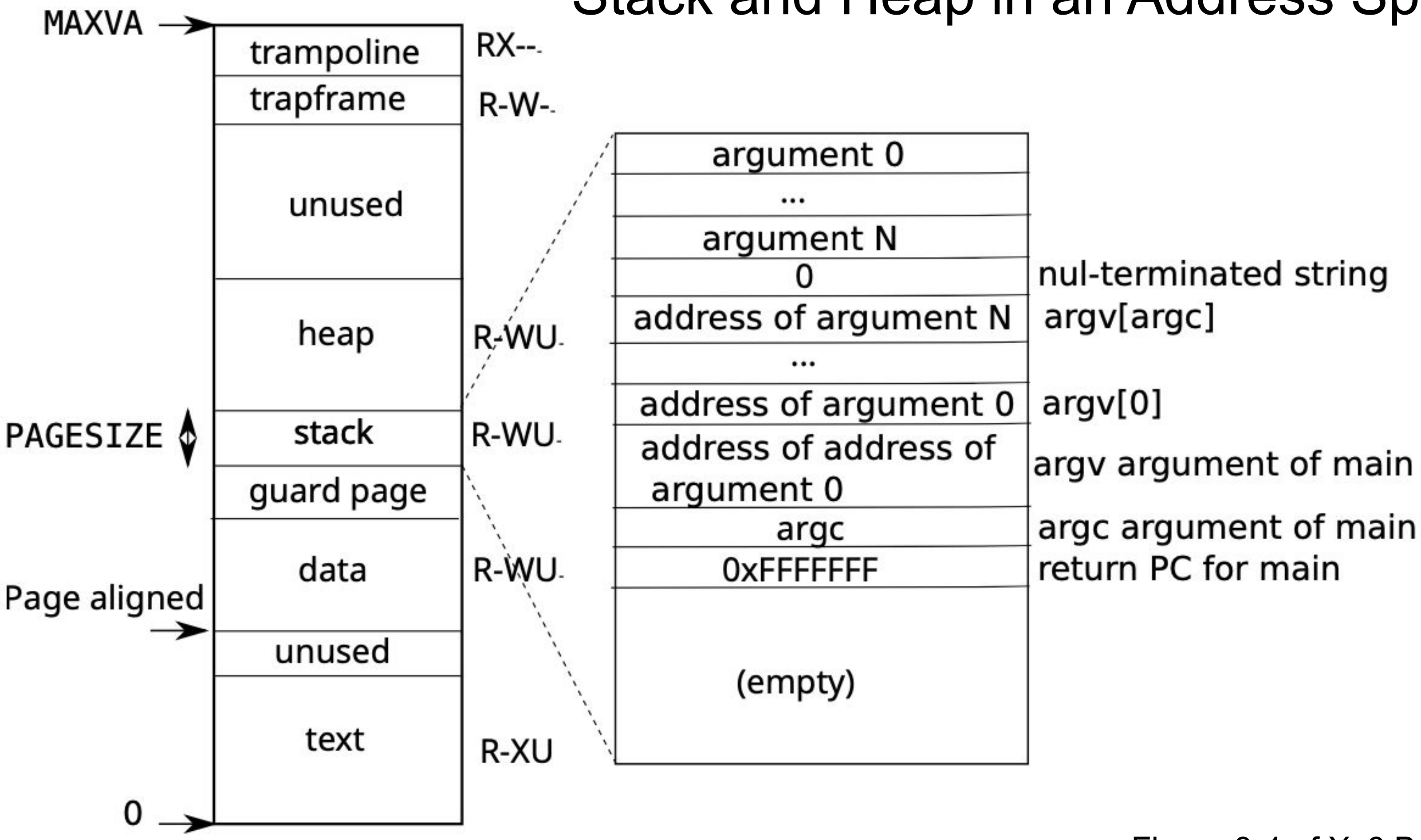


Figure 3-4 of Xv6 Book

Common Memory Problems

- Using memory after freeing it
- Freeing the same object more than once
- Forgetting to initialize memory (nothing is zeroed automatically)
- Writing beyond the end of an array (buffer overflow)
- Forgetting to free an object (memory leak)
- Casting an object to the wrong type
- Forgetting to check if an allocation failed
- Using pointers to locations on the stack (if they could return)

Why use C for an OS

- Good for low-level programming
  - Can manipulate address spaces directly without language abstractions
  - Easy to access hardware structures and RISC-V instructions
- Kernel is in complete control of memory allocation
  - In fact, you can build a memory allocator using C
  - No garbage collection
- Efficient and fast: compiled, no interpreter
- Why not?
  - Easy to write incorrect/insecure code!
  - Limited abstractions.

Primitive Types - RISC-V

- char: 1 byte
- short: 2 bytes
- int: 4 bytes
- long: 8 bytes
- long long: 8 bytes
- void *: 8 bytes (any pointer type is this size)
- Qualifiers: unsigned (nonnegative), const (can’t be modified), static (only accessed within the file)
- sizeof(type) returns the size of a type
-

Using C typedefs

- xv6 uses typedefs to make the size of types more obvious
- typedef unsigned char uint8; // uint8 is the same as unsigned char
- typedef unsigned short uint16;
- typedef unsigned int uint32;
- typedef unsigned long uint64;

C structs

```
struct a {
  int foo;
};
struct b {
   struct a bar;
   long baz;
};
```

Q: What will printf(“%ld”, sizeof(struct b)) print?

Casting

- Converts one type to another
- Example:
  - int foo = 10;
  - long bar = (long)foo;

Pointer Arithmetic

```
void foo(void *ptr)
{
  void *pos = ptr + 10;          // doesn't compile!
  void *pos = (char *)ptr + 10; // works fine
  uint64 addr = (uint64)pos;    // can convert to int
  addr += 10;
  pos = (void *)addr;           // and back again
}
```

Bitwise Operators

0b10001 & 0b10000 == 0b10000

0b10001 | 0b10000 == 0b10001

0b10001 ^ 0b10000 == 0b00001

~0b1000 == 0b0111

Arrays

```
int foo[5];
int i;
for (i = 0; i < 5; i++) {
foo[i] = i; }
// now foo contains 0, 1, 2, 3, 4
```

What does this print?

```
#include <stdio.h>
int main() {
  int x[5];
  printf("%p\n", x);   // equivalent to &x[0]
  printf("%p\n", x+1); // equivalent to &x[0] + 1
  printf("%p\n", &x);  // pointer to x
  printf("%p\n", &x+1);// eqv. to x + sizeof(x[5])
  return 0;
}
```

Source: https://blogs.oracle.com/linux/post/the-ksplice-pointer-challenge

Conclusion

- Many layers of abstraction in memory
- Writing an OS requires you to be aware of all of them
- C is a low-level language, so it’s good at doing this
- But many pitfalls; large potential for bugs and security problems