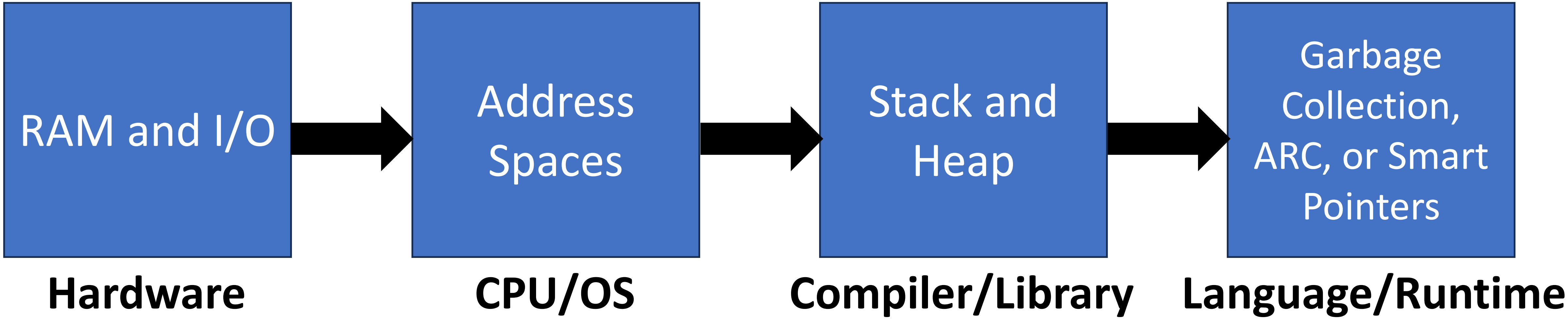
Programming xv6 in C

Adam Belay <abelay@mit.edu>

Today’s agenda

- 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’s many layers of abstraction



RAM and I/O

Address Spaces

Stack and Heap

Garbage Collection, ARC, or Smart Pointers

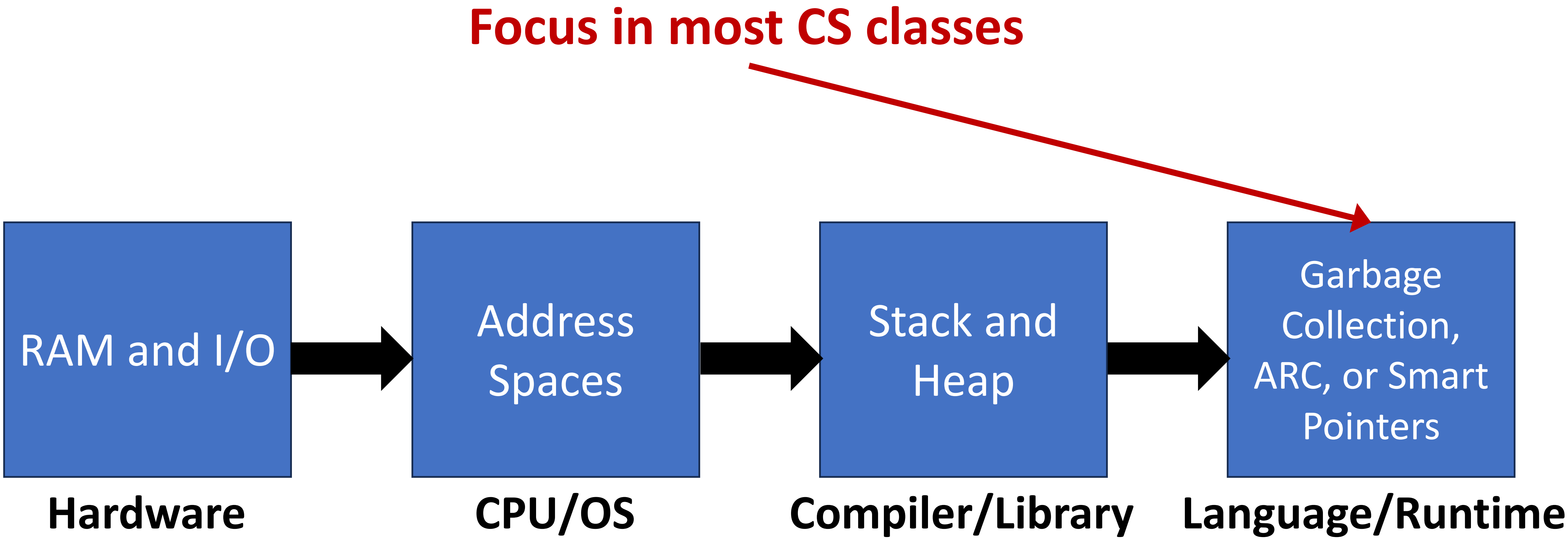
Hardware

CPU/OS

Compiler/Library

Language/Runtime

Memory's many layers of abstraction



Focus in most CS classes

RAM and I/O

Address Spaces

Stack and Heap

Garbage Collection, ARC, or Smart Pointers

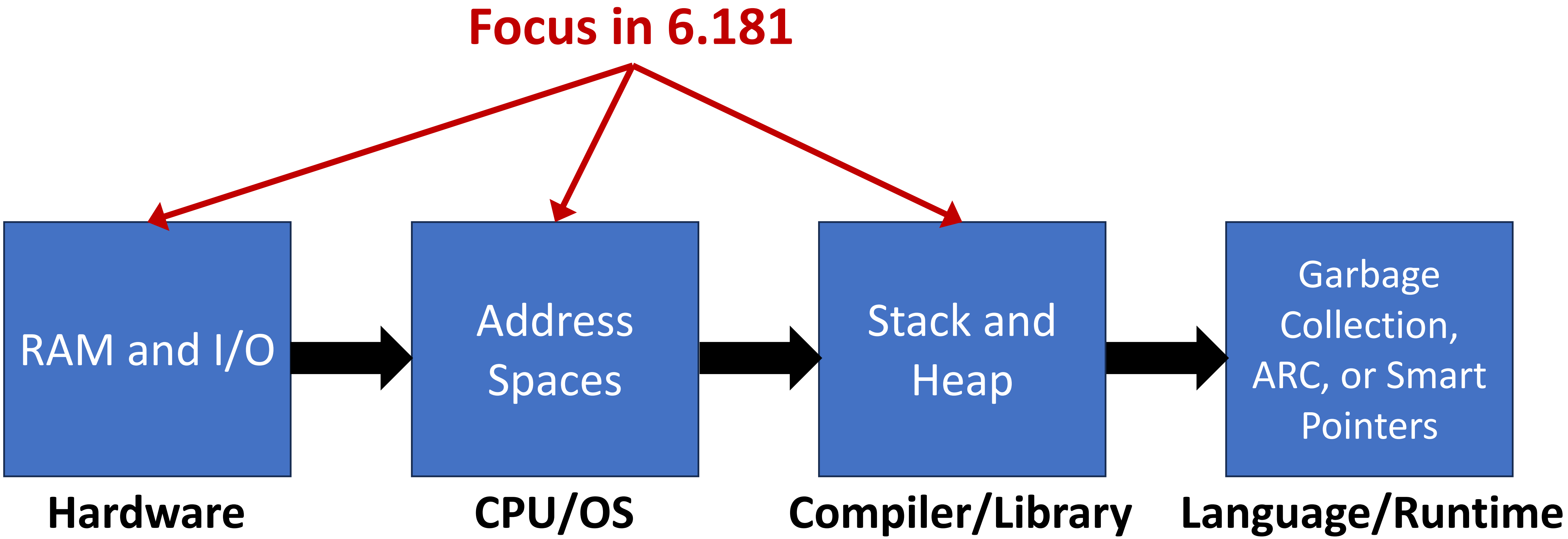
Hardware

CPU/OS

Compiler/Library

Language/Runtime

Memory’s many layers of abstraction



Focus in 6.181

RAM and I/O

Address Spaces

Stack and Heap

Garbage Collection, ARC, or Smart Pointers

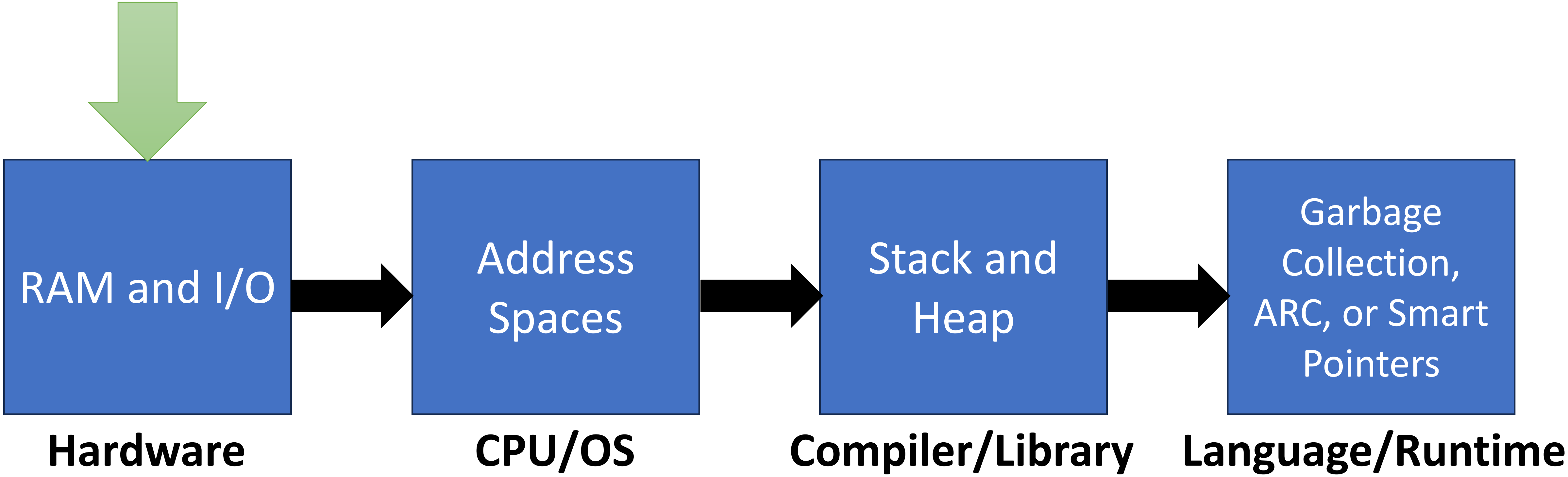
Hardware

CPU/OS

Compiler/Library

Language/Runtime

Memory’s many layers of abstraction



RAM and I/O

Address Spaces

Stack and Heap

Garbage Collection, ARC, or Smart Pointers

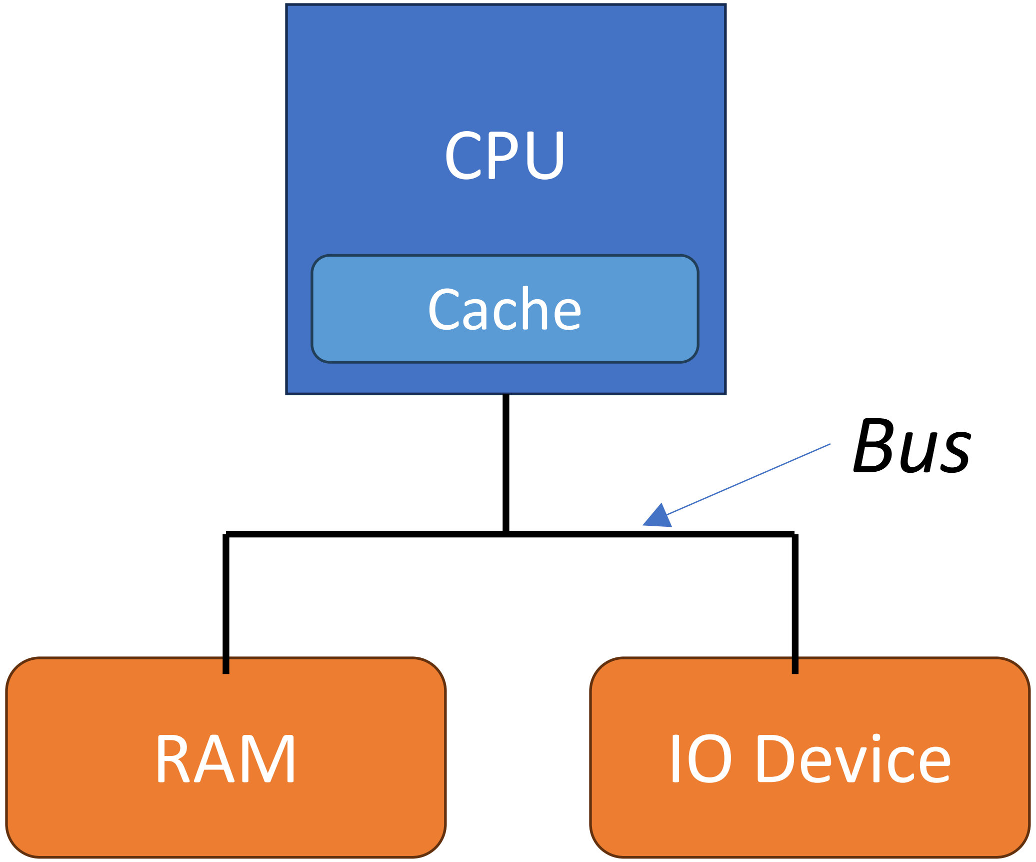
Hardware

CPU/OS

Compiler/Library

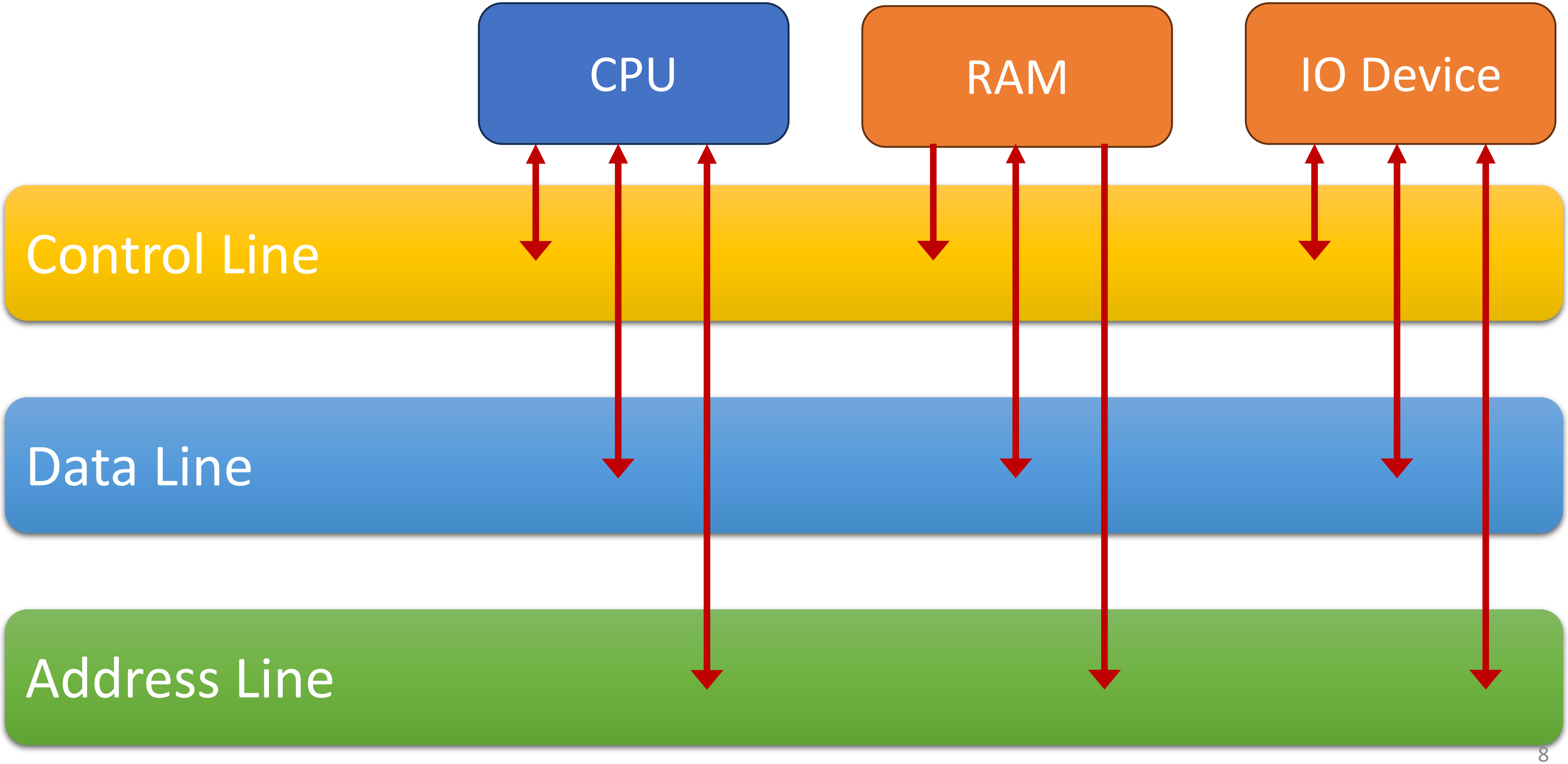
Language/Runtime

Hardware layer: RAM and I/O

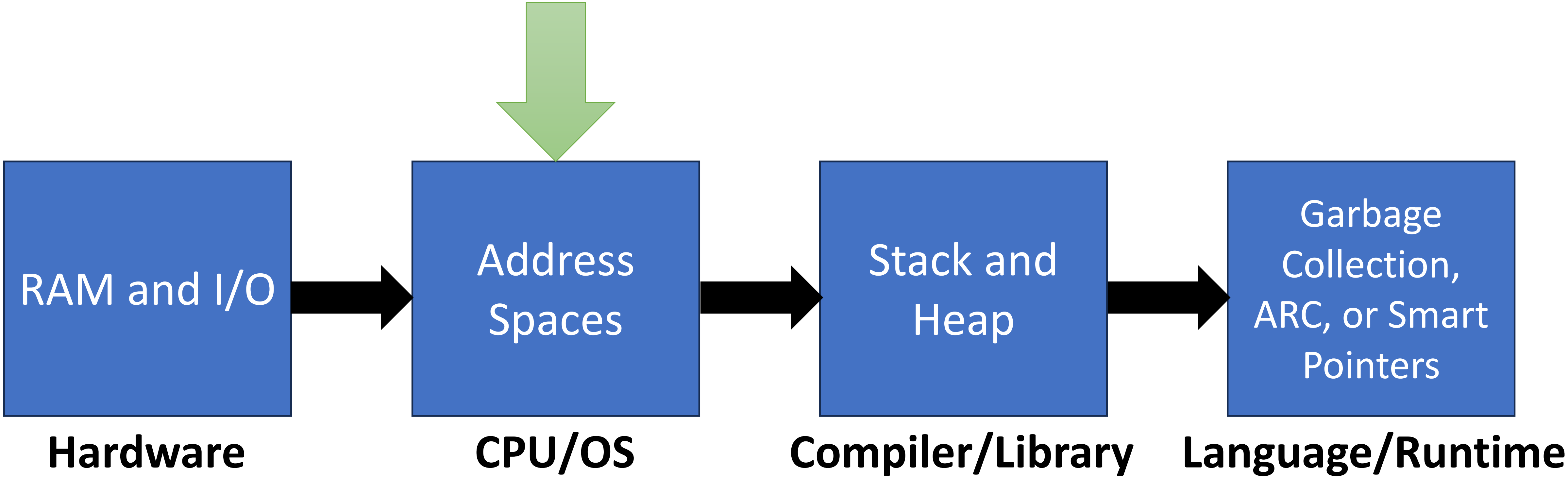


- A bus transfers data between components in the computer
- A cache remembers data previously fetched from the bus
- Speeds up the CPU by reducing the number of bus accesses
- Q: What is an IO Device?

How does a bus work?



Memory’s many layers of abstraction



RAM and I/O

Address Spaces

Stack and Heap

Garbage Collection, ARC, or Smart Pointers

Hardware

CPU/OS

Compiler/Library

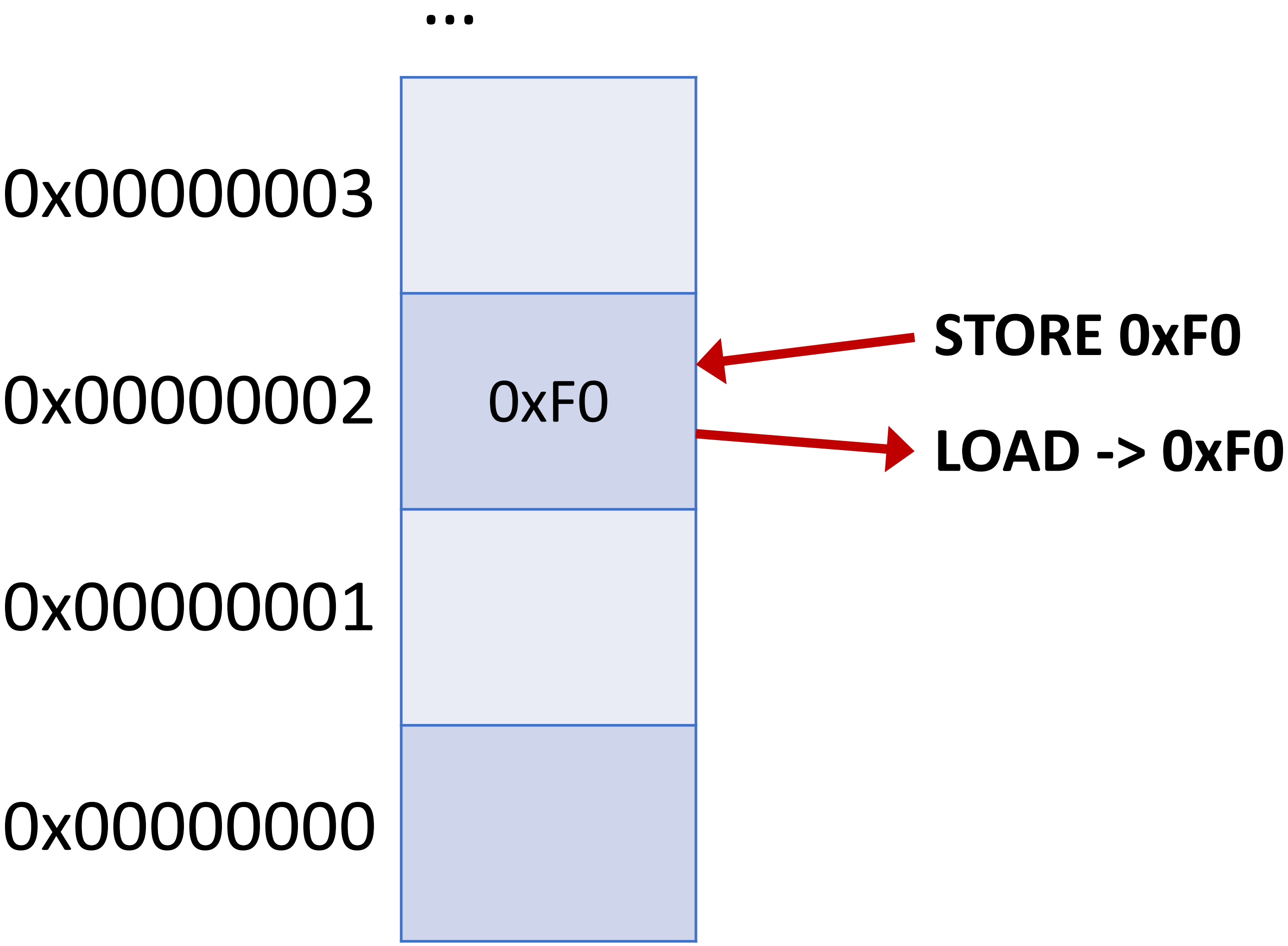
Language/Runtime

CPU/OS layer: Address Spaces

- Problem: Bus interface is too low-level to do anything useful!
- Idea: Represent bus as a giant array of data
- This is called an ***address space***
- Each array element is a byte (8 bits)

The address is the array index in bytes!

How to interact with an address space?



Idea #1: Address spaces can have holes

…

0x00000003

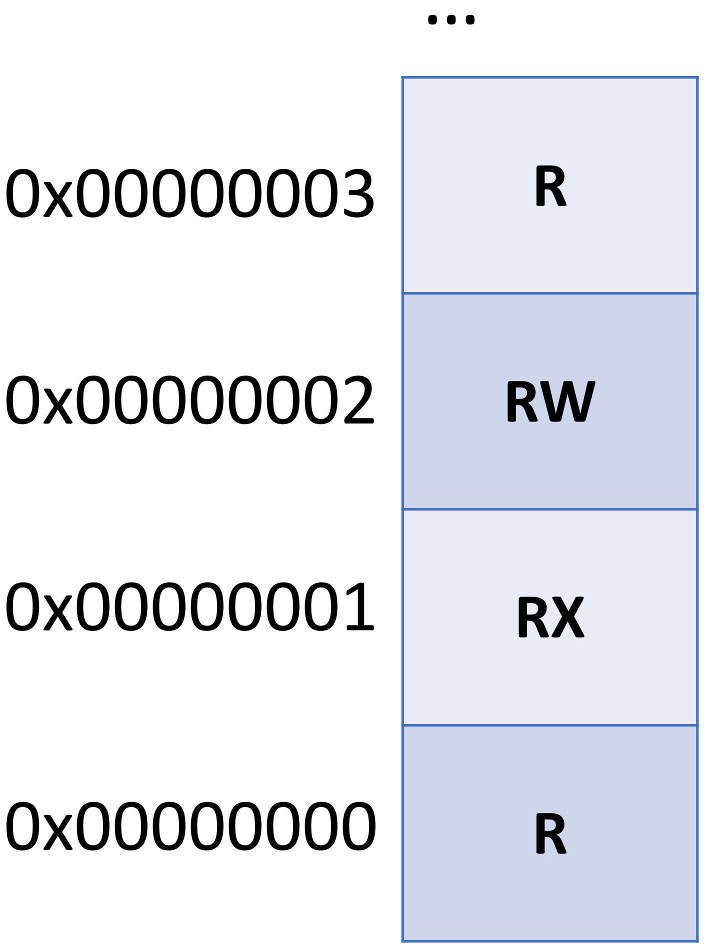
0x00000002

0x00000001

0x00000000

- Usually address space is much larger than RAM
- Addresses that can be accessed are referred to as “mapped”
- And holes that can’t be accessed are “unmapped”
- Q: What happens if the CPU loads or stores to an unmapped region?

Idea #2: Address spaces can have permissions



…

0x00000003	**R**
0x00000002	**RW**
0x00000001	**RX**
0x00000000	**R**

- Read (R) -> Can load
- Write (W) -> Can store
- Execute (X) -> Can execute as code

- Q: Why have permissions?
- Q: What happens if the CPU loads or stores an address without permission?

Idea #3: Combine RAM and devices

	…
0x00000003	**Memory (code)**
0x00000002	**Memory (data)**
0x00000001	**IO Device**
0x00000000	**Memory (data)**

- Not as obvious as it sounds; e.g., x86 originally put I/O in a separate address space from memory
- Programmer can then interact with IO devices through loads and stores!
- Treating code and data the same (as memory) is also a powerful idea, called a *Von Neumann architecture*.

More ideas not discussed today

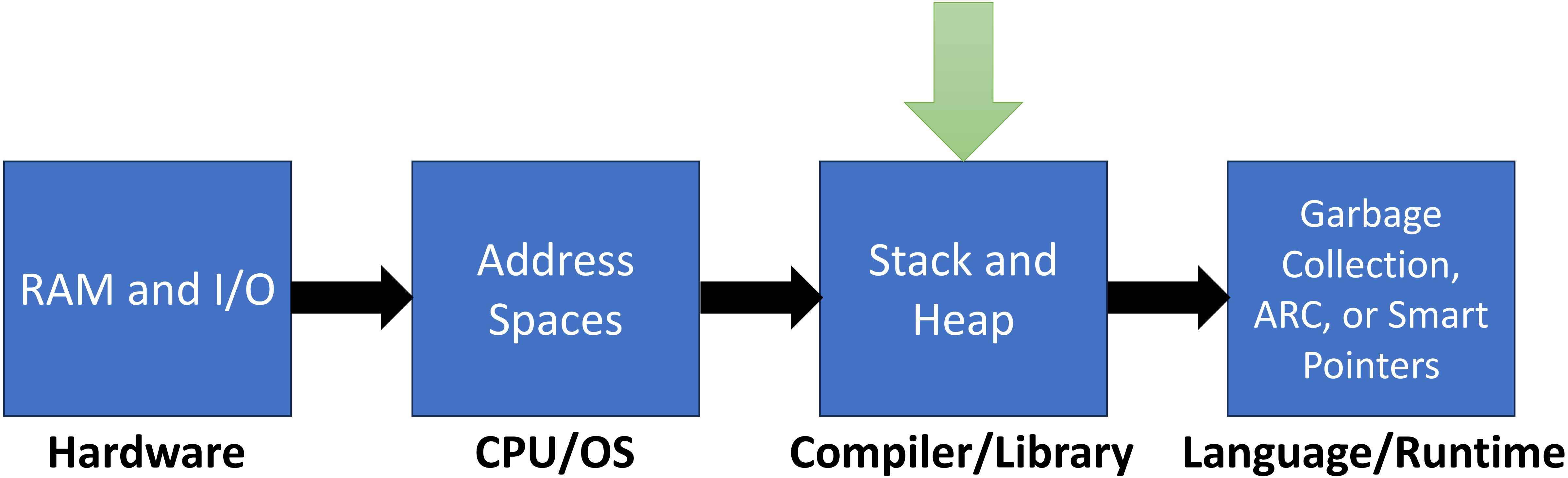
Typical granularity for mappings is a page (4KB), not a byte

- **Idea #4:** Virtual memory
  - Allows each process to have its own address space

- **Idea #5:** Cache coherence and consistency
  - Allows multiple CPUs to share memory in an address space

- Will be covered in later lectures

Memory’s many layers of abstraction



RAM and I/O

Hardware

Address Spaces

CPU/OS

Stack and Heap

Compiler/Library

Garbage Collection, ARC, or Smart Pointers

Language/Runtime

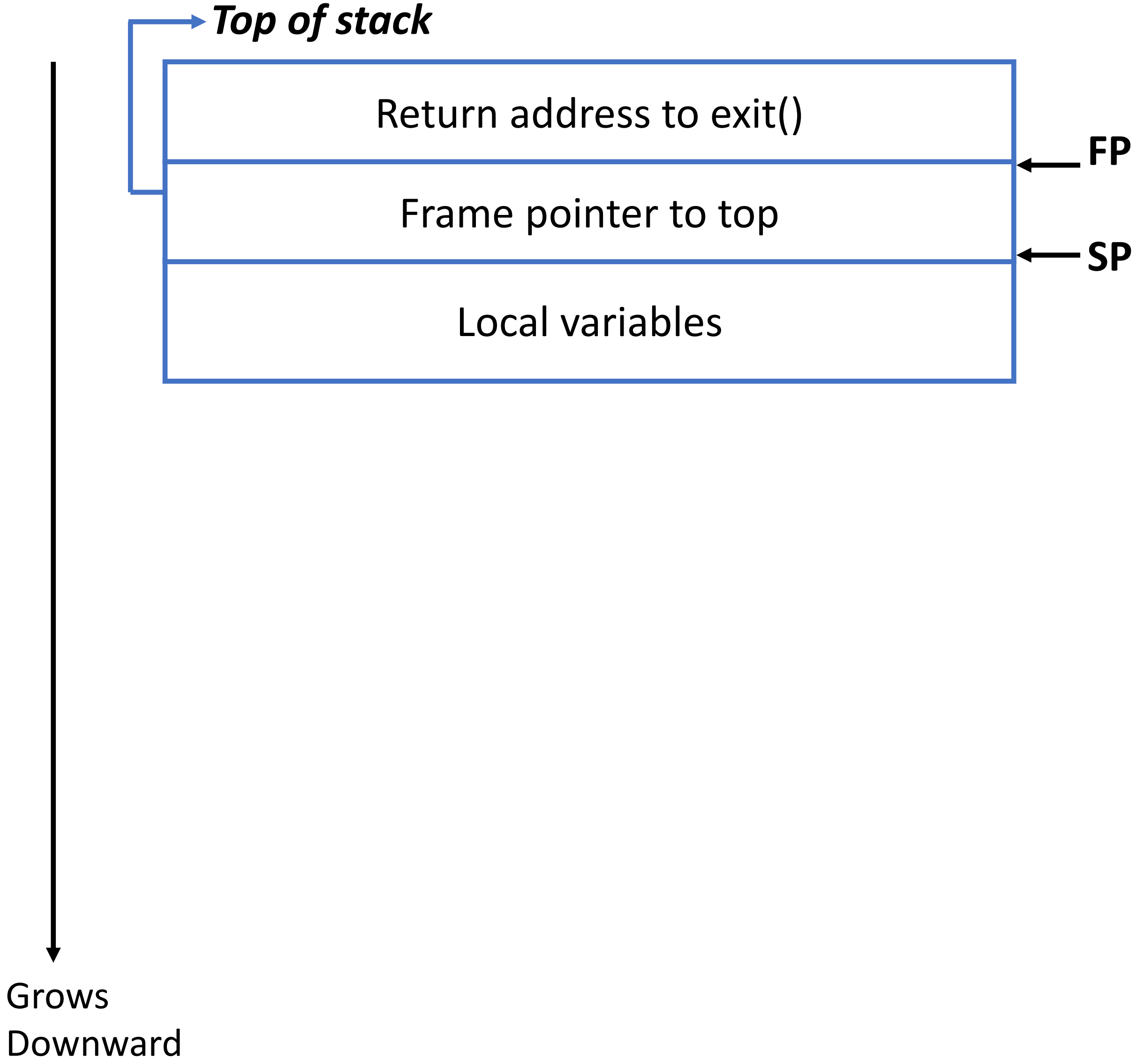
Compiler/Library Layer: Stacks and Heaps

- Problem: An address space is also too low-level!
- How can we decide where in the array to store things?
- This problem is called ***memory allocation***

- Two basic approaches:

1. A ***stack*** allocates memory when a function is called and frees it when a function returns
2. A ***heap*** manages memory that is allocated and freed independently of function invocations

Stack basics



Top of stack

Return address to exit()

Frame pointer to top

Local variables

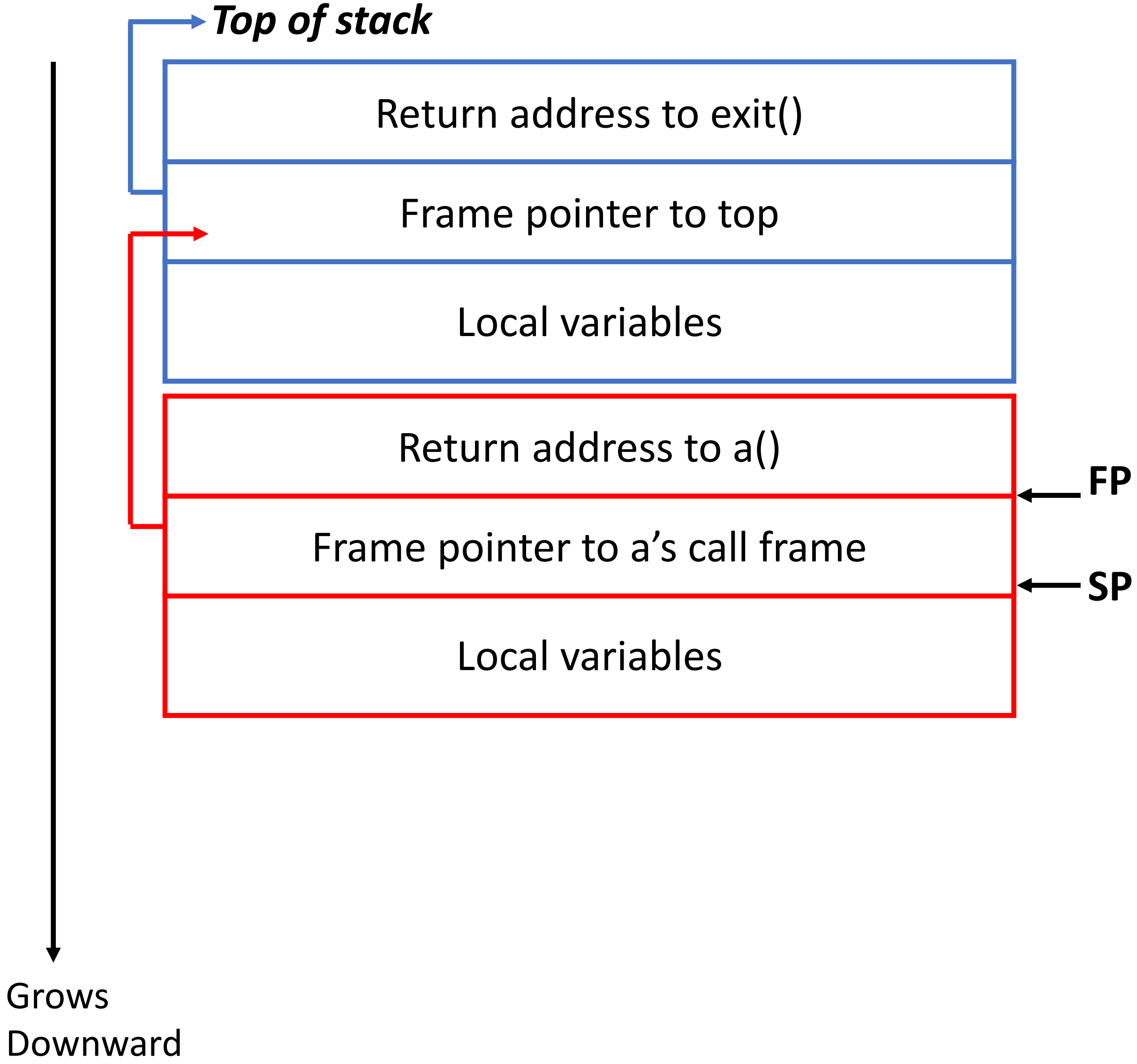
FP

SP

Grows
Downward

a(args...)

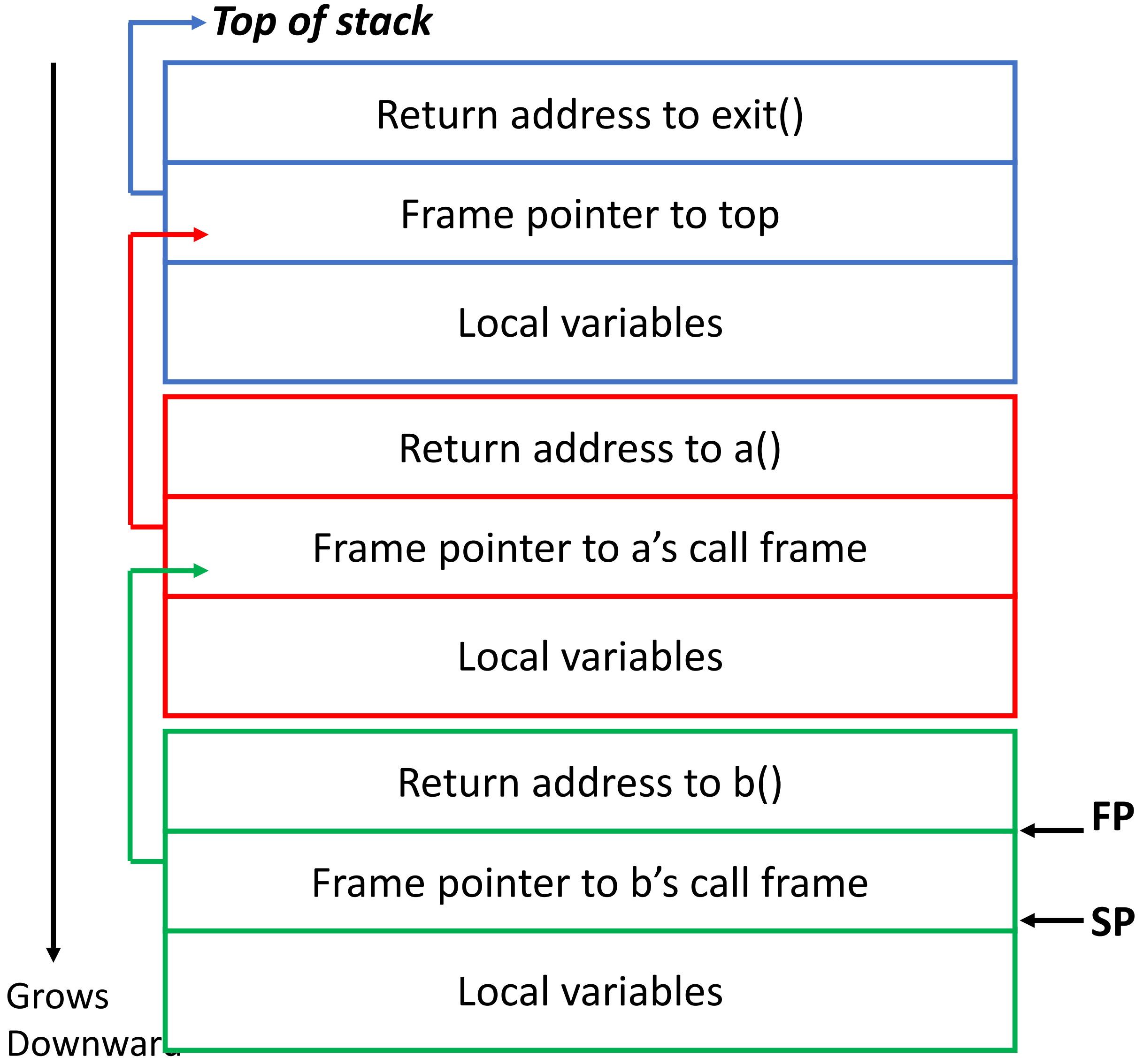
Stack basics



a(args...)

↳**b(args...)**

Stack basics



Top of stack

Return address to exit()

Frame pointer to top

Local variables

Return address to a()

Frame pointer to a's call frame

Local variables

Return address to b()

Frame pointer to b's call frame

Local variables

FP

SP

Grows
Downward

a(args…)
↳b(args…)
  ↳c(args…)

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 a heap

Example:

```
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);
```

Building a heap allocator

- Problem: Need to keep track of what regions are free and allocated in an array of memory (the heap)
- Turns out to be an interesting area of research even today
- Many design tensions; best solution depends on the allocation pattern

Q: When is it better to use a stack vs. a heap?

Q: When is it better to use a stack vs. a heap?

- Always prefer a stack, except if the object must remain valid after the function returns or if the object is too large!
- Why? More efficient and simpler
- Note: A stack is generally much smaller than the heap

Tying the stack and heap to an address space

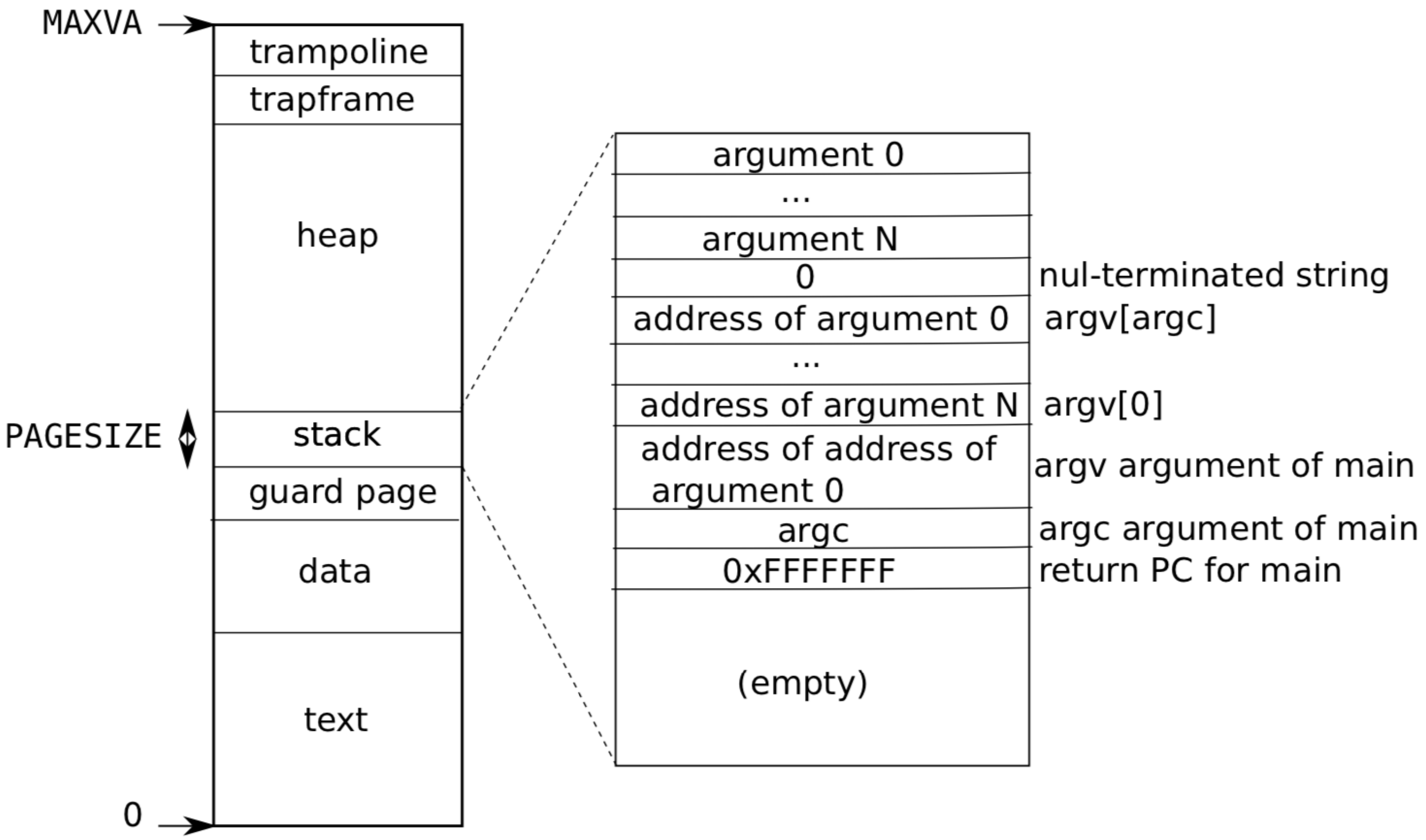


Figure 3.4: Memory layout of a user process with its initial stack.

Common memory management pitfalls

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

Why build an OS in C?

- 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 (RISCV-64)

- 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

Typedef declares aliases

```
// example: xv6 uses these 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;
```

Structs combine together types

```
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 operations

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);
  printf("%p\n", x+1);
  printf("%p\n", &x);
  printf("%p\n", &x+1);
  return 0;
}
```

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

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