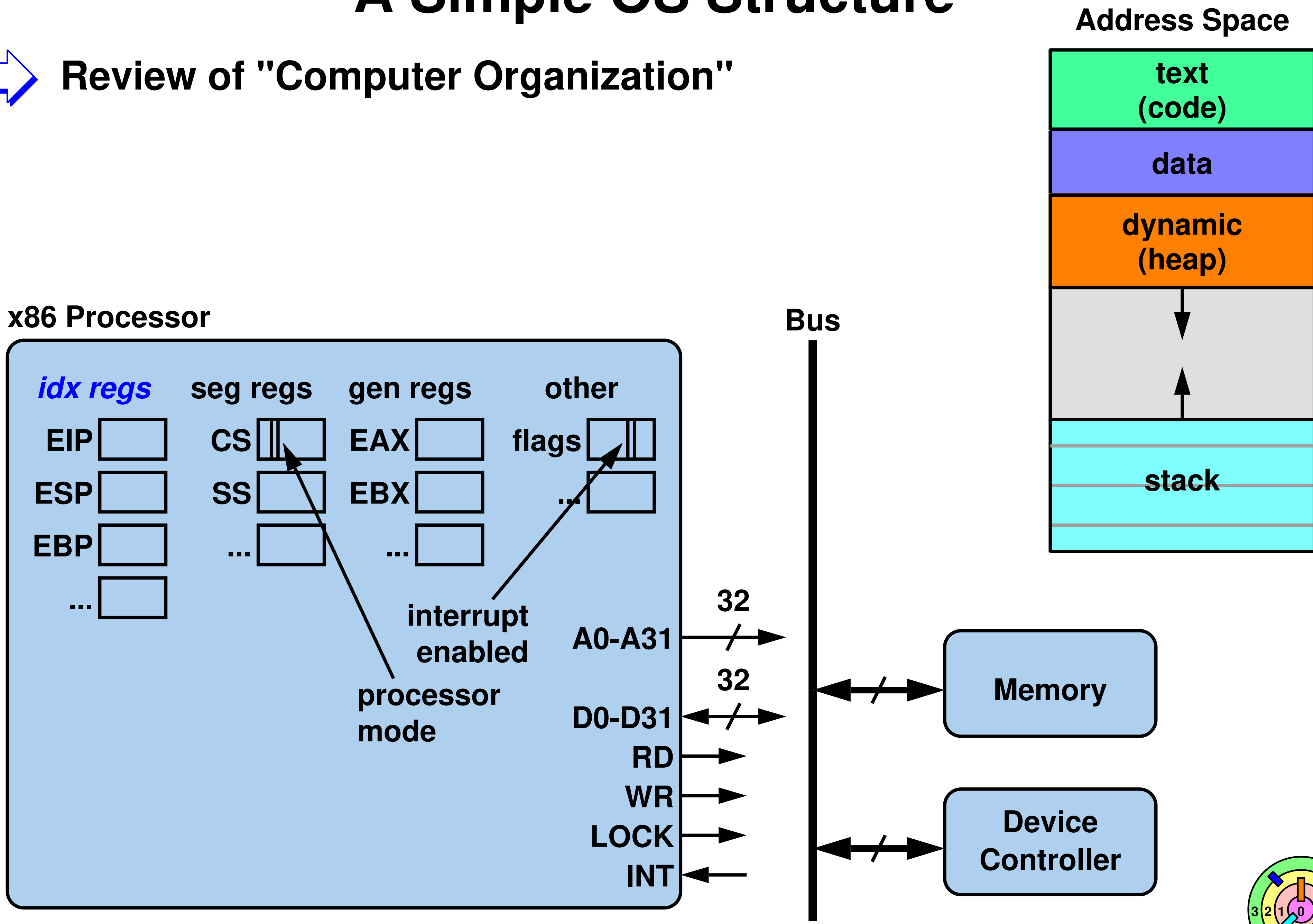
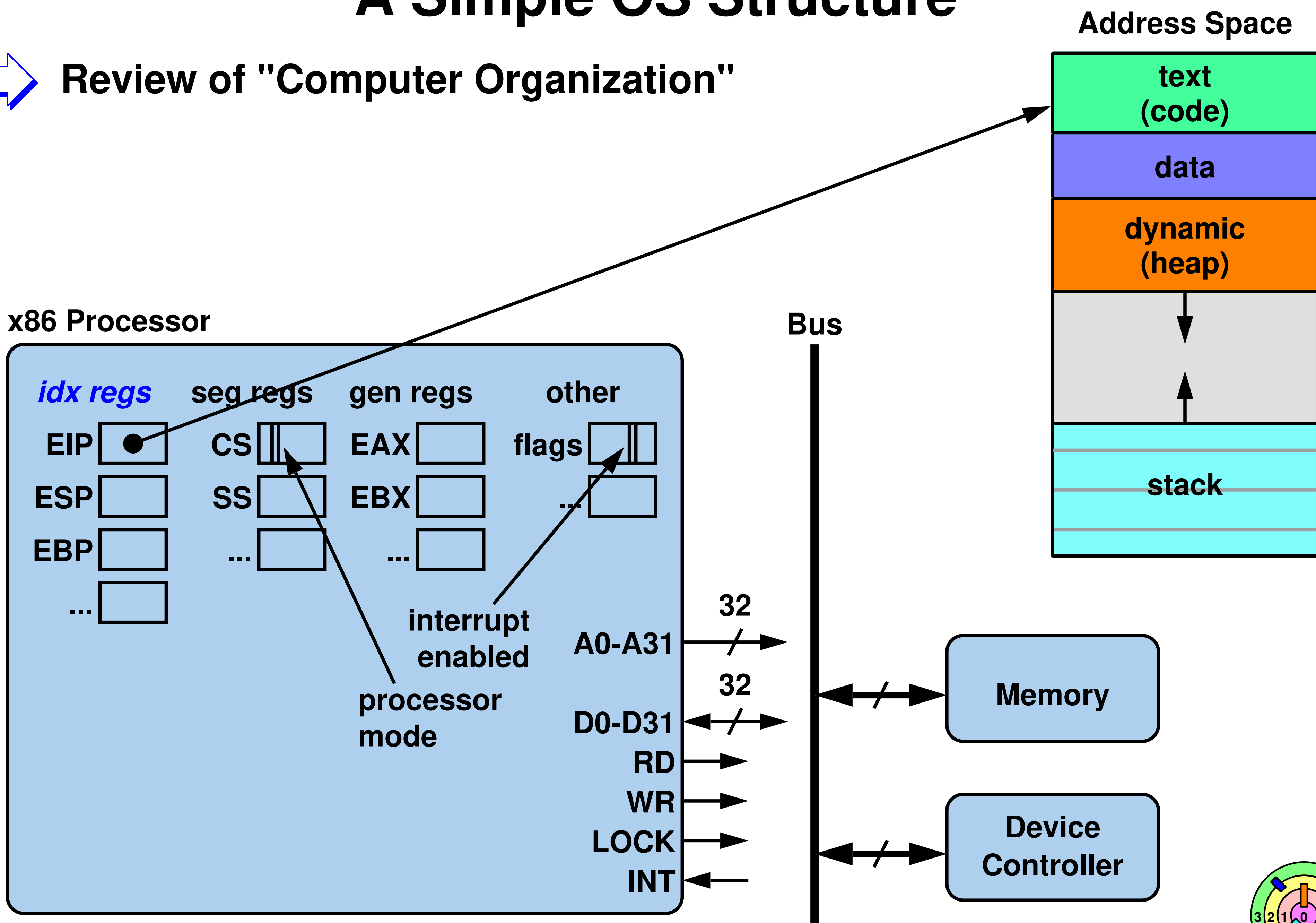
# A Simple OS Structure

- Review of "Computer Organization"



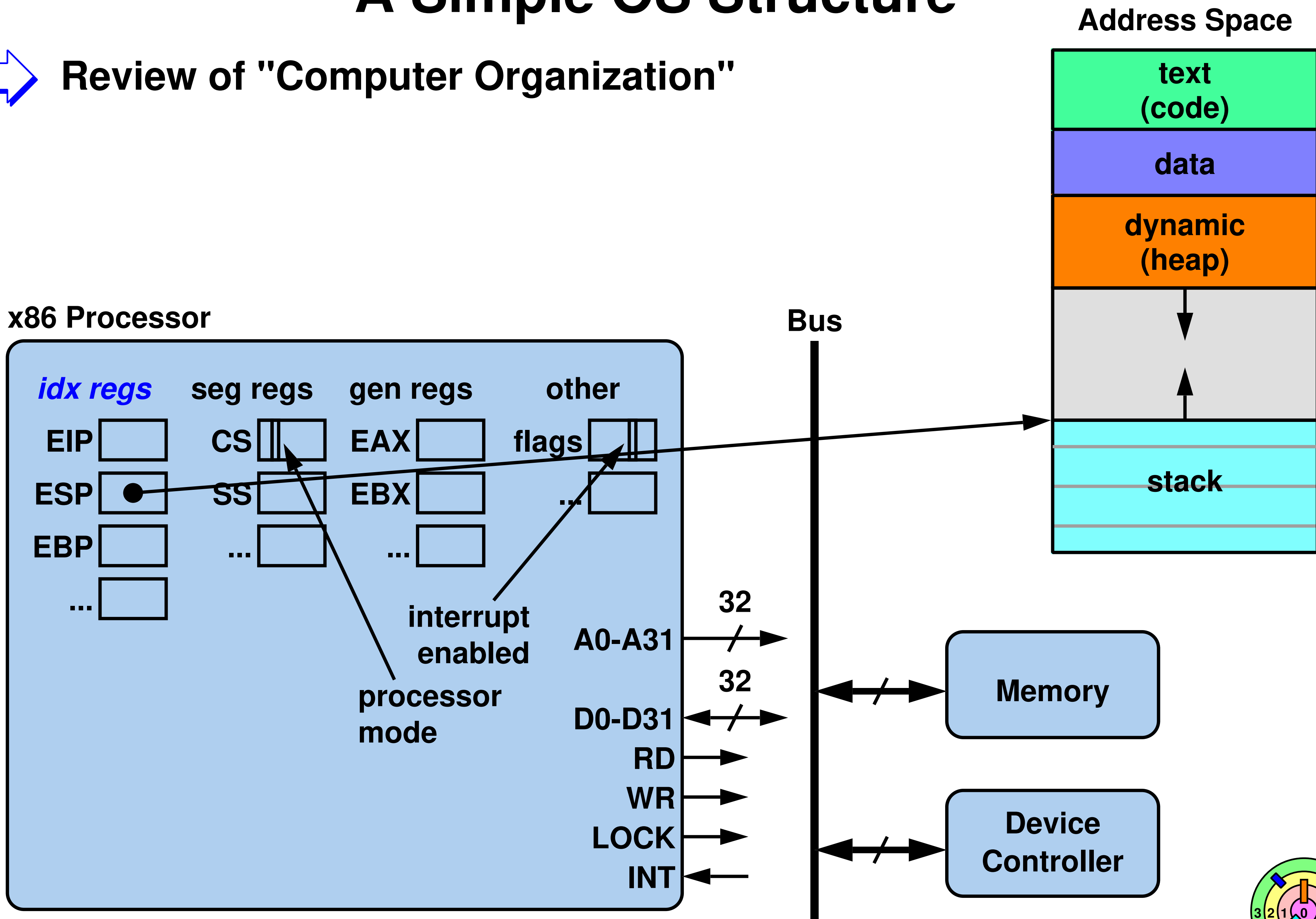
# A Simple OS Structure

- Review of "Computer Organization"



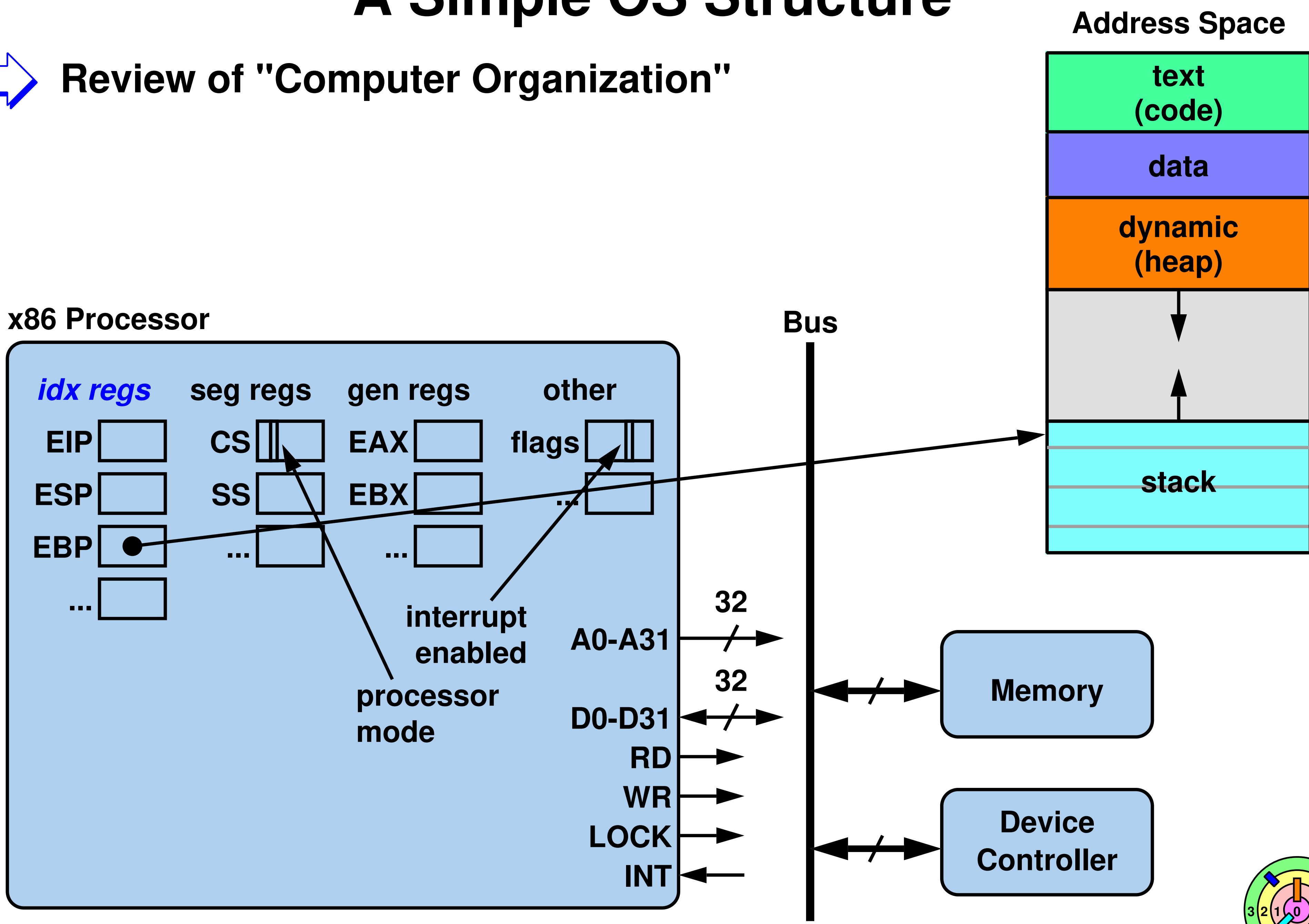
# A Simple OS Structure

- Review of "Computer Organization"



# A Simple OS Structure

- Review of "Computer Organization"

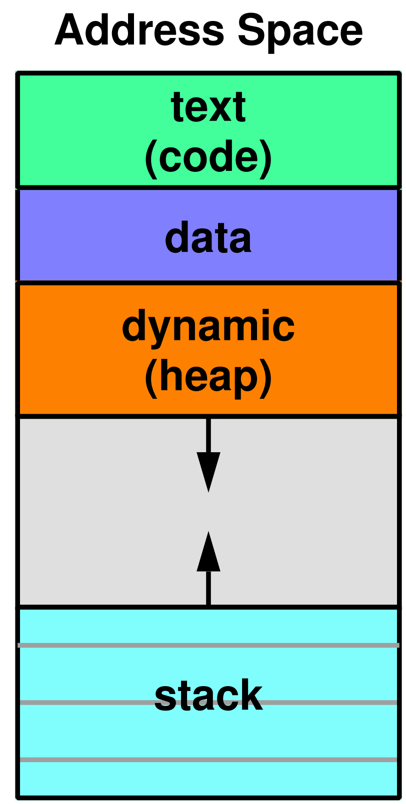


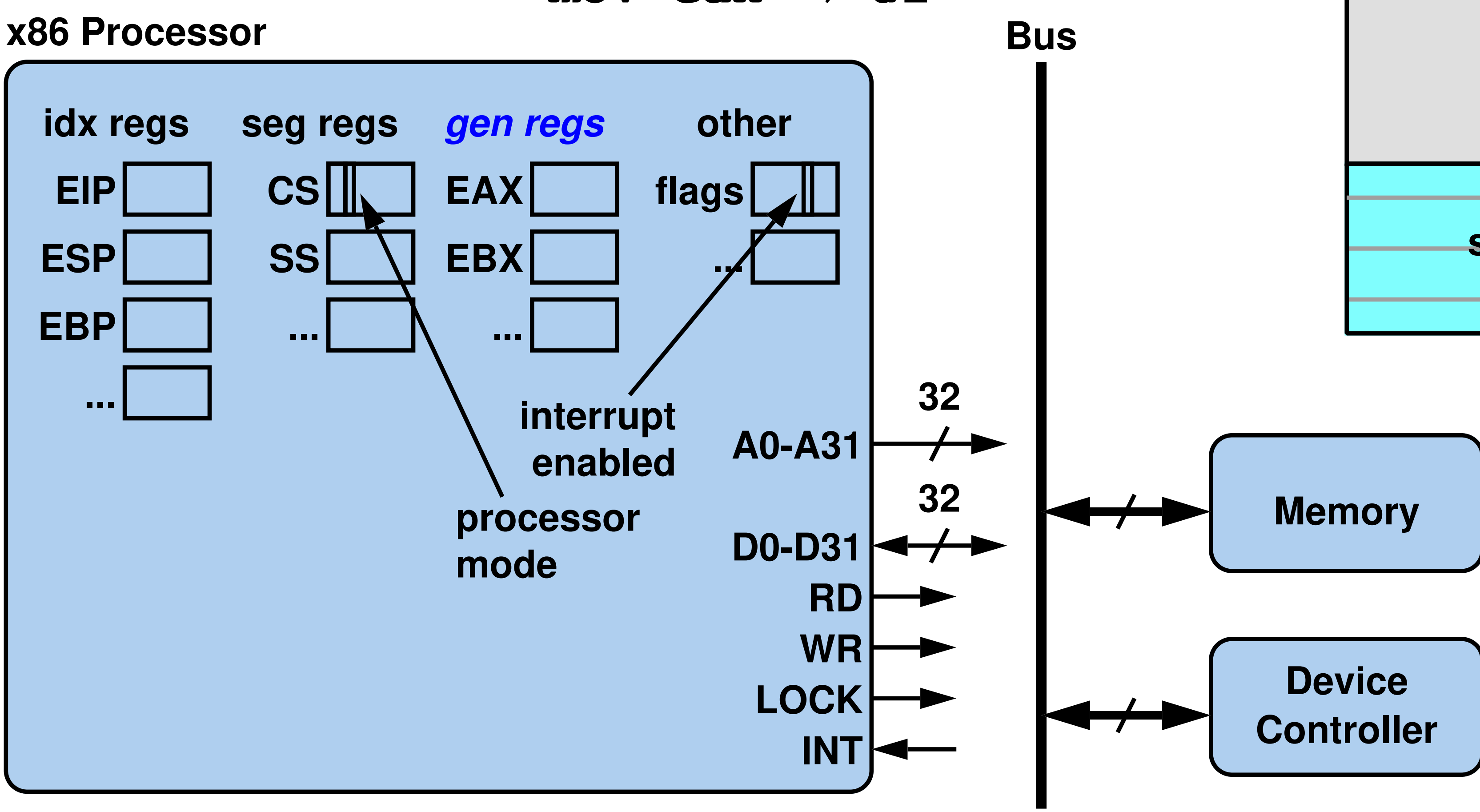
# A Simple OS Structure

## ⇨ Review of "Computer Organization"

z = x + y ⇨

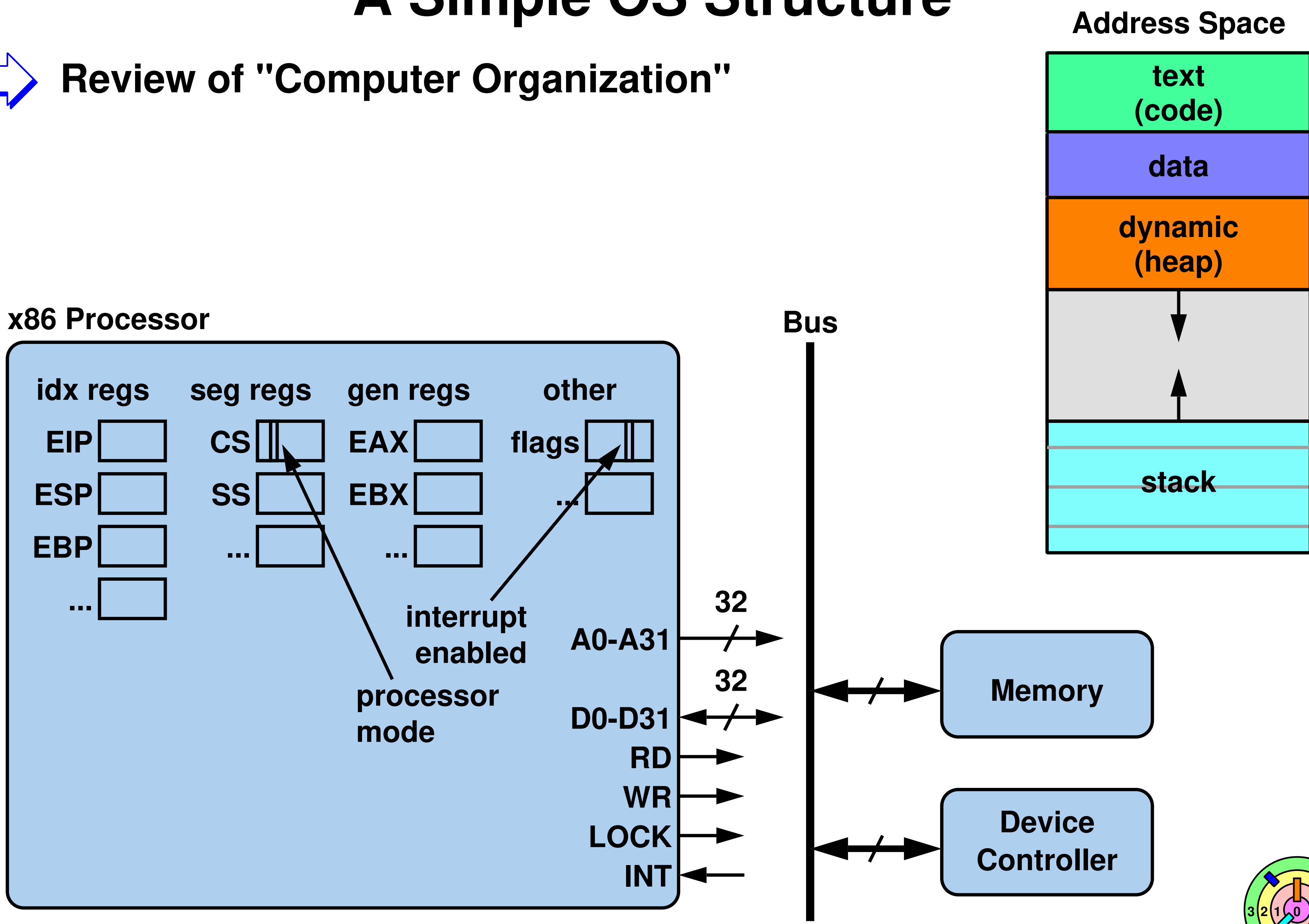
```
mov &x → eax
mov &y → ebx
add(eax,ebx)
mov eax → &z
```



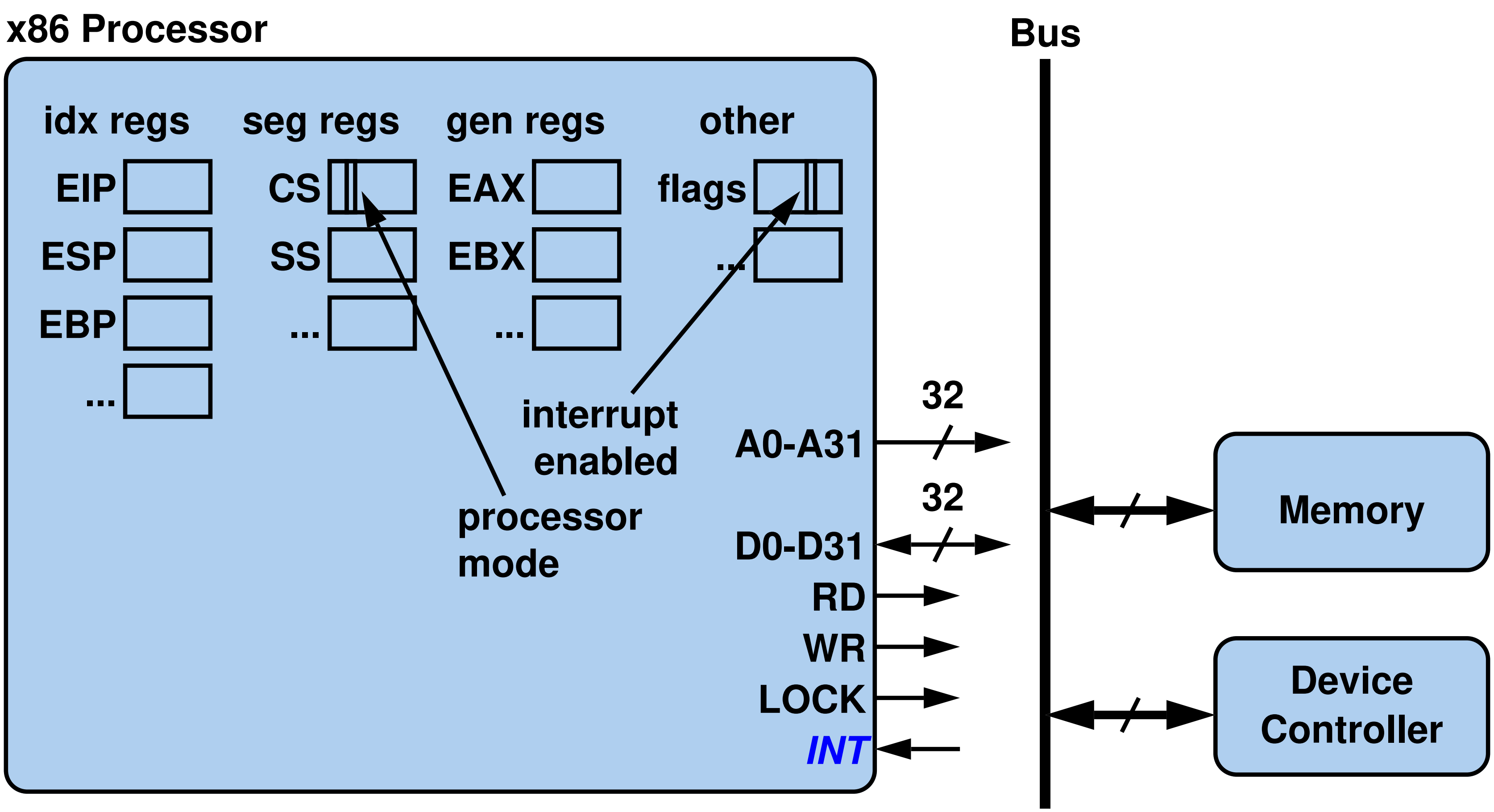
# A Simple OS Structure

- Review of "Computer Organization"

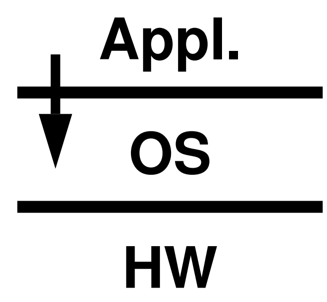


# A Simple OS Structure

- Some important terms:
  - *interrupt pending*
  - *interrupt delivery*
  - *interrupt context*
  - *thread context*



# Traps




- ***Traps*** **are the general means for invoking the kernel from user code**
  - **although we usually think of traps as *errors***
    - **divide by zero, segmentation fault, bus error, etc.**
  - **but they don't have to be**
    - ***system calls*, *page fault*, etc.**

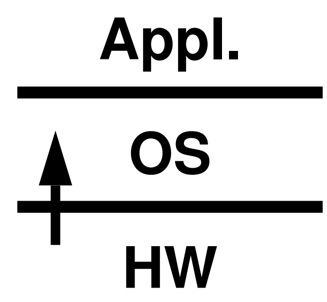
- **Traps always elicit some sort of response**
  - **for programming errors, the default action is to *terminate* the user program**
  - **for system calls, the OS is asked to perform some service**
  - **for page faults, the OS need to fix the virtual memory map**

# A Special Kind Of Trap - System Calls

- Invoking OS functionality in the kernel is more complex
  - but we want to make it look simple to applications
  - must be done carefully and correctly
    - really cannot trust the application programmers to do the right thing every time

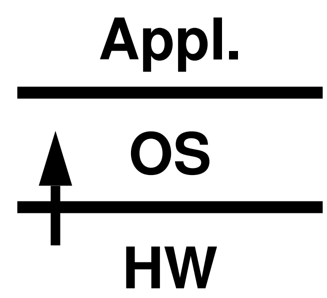
- Provide *system calls* through which user code can access the kernel *in a controlled manner*
  - any necessary checking on whether the request should be permitted can be done in the system call
    - all done in user mode
  - if all goes well
    - sets things up
    - *traps* into the kernel by executing a special machine instruction, i.e., the "trap" machine instruction
    - the kernel figures out why it was invoked and handles the trap
  - more in Ch 3

# Interrupts




- An *interrupt* is a request from an *external device* for a response from the *processor*
  - most *hardware interrupts* are *I/O completion interrupts*
    - an I/O device is telling the CPU, "I am done" (and "what do you want me to do next?")
    - I/O devices are also hardware, they can run in parallel with the CPU, don't keep them idle unless you have nothing for them to work on
  - *interrupts* are handled *independently of any user program*
    - unlike a *trap*, which is handled *as part of the program* that caused the trap where response to a trap directly affects that program
  - response to an interrupt may or may not indirectly affect the currently running program
    - often has *no direct effect* on the currently running program

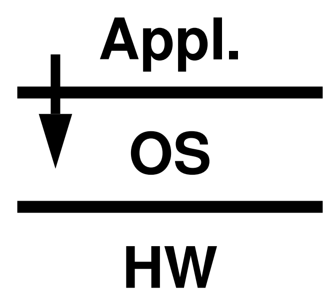
# Interrupts




- An *interrupt* is an *asynchronous* event
  - it's asynchronous with respect to the executing entity (threads or OS)

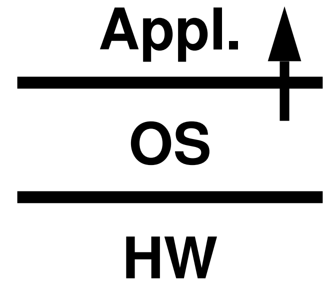
- A *trap* occurs *synchronously* with respect to the executing entity
  - when your thread executes a divide-by-zero instruction, we know exactly where it happens and we know when it will happen

# Software Interrupt




- There's also something called ***software interrupt***
  - generated programmatically (i.e., not by a device) when executing a machine instruction
    - e.g., executing an "interrupt" machine instruction
    - x86 CPU uses a software interrupt (i.e., "`int 0x2e`") to implement the "trap" machine instruction
      - other CPUs may have a separate "trap" machine instruction
  - this is very different from a hardware interrupt
    - although the mechanisms of handling interrupts are all very similar as we will see in Ch 3

# Upcall




- A program may establish a handler *(i.e., a signal handler)* to be invoked in response to the error
  - the handler might clean up after the error and then terminate the program, or it might perform corrective action and continue with normal execution
  - more in Ch 2

- The *upcall* mechanism
  - *signals* allow the kernel to invoke code that's part of user program
    - for example, you can set a timer to expire at a certain time, when it expires, the OS can use the upcall mechanism to call a specified user function on behalf of the user program

# 1.3 A Simple OS

- OS Structure
- *Processes, Address Spaces, & Threads*
- Managing Processes
- Loading Program Into Processes
- Files

# Program Execution

- Fundamental *abstraction* of *program execution*
  - memory
    - address space
      - things that are addressable by the program are kept together here
    - in Sixth-Edition Unix, processes do not share address space
    - recall that *process* is an abstraction of *memory*
  - processor(s)
    - recall that *thread* is an abstraction of *processor*
  - "execution context"
    - which represents the *state* of a process and its threads
    - represents exactly "where you are" in the program
    - a thread needs some sort of a context to execute

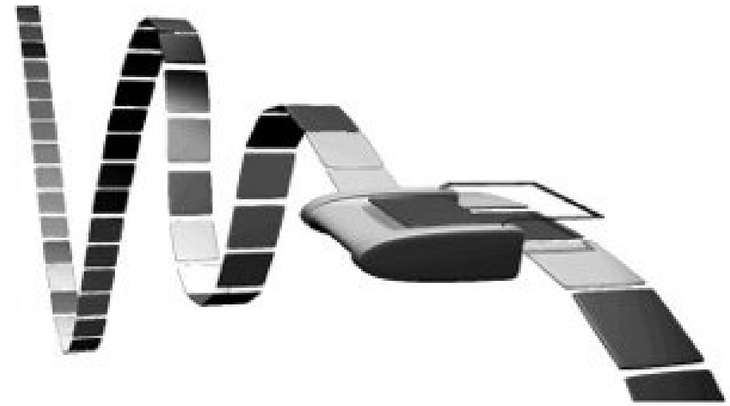
- Note: multiple meanings of the word "context" in this class
  - *save (execution) context* and *restore (execution) context*
  - *thread context* vs. *interrupt context*

# A Program

```
const int nprimes = 100;
int prime[nprimes];
int main() {
  int i;
  int current = 2;
  prime[0] = current;
  for (i=1; i<nprimes; i++) {
     int j;
  NewCandidate:
     current++;
     for (j=0; prime[j]*prime[j] <= current; j++) {
        if (current % prime[j] == 0)
           goto NewCandidate;
     }
     prime[i] = current;
  }
  return(0);
}
```

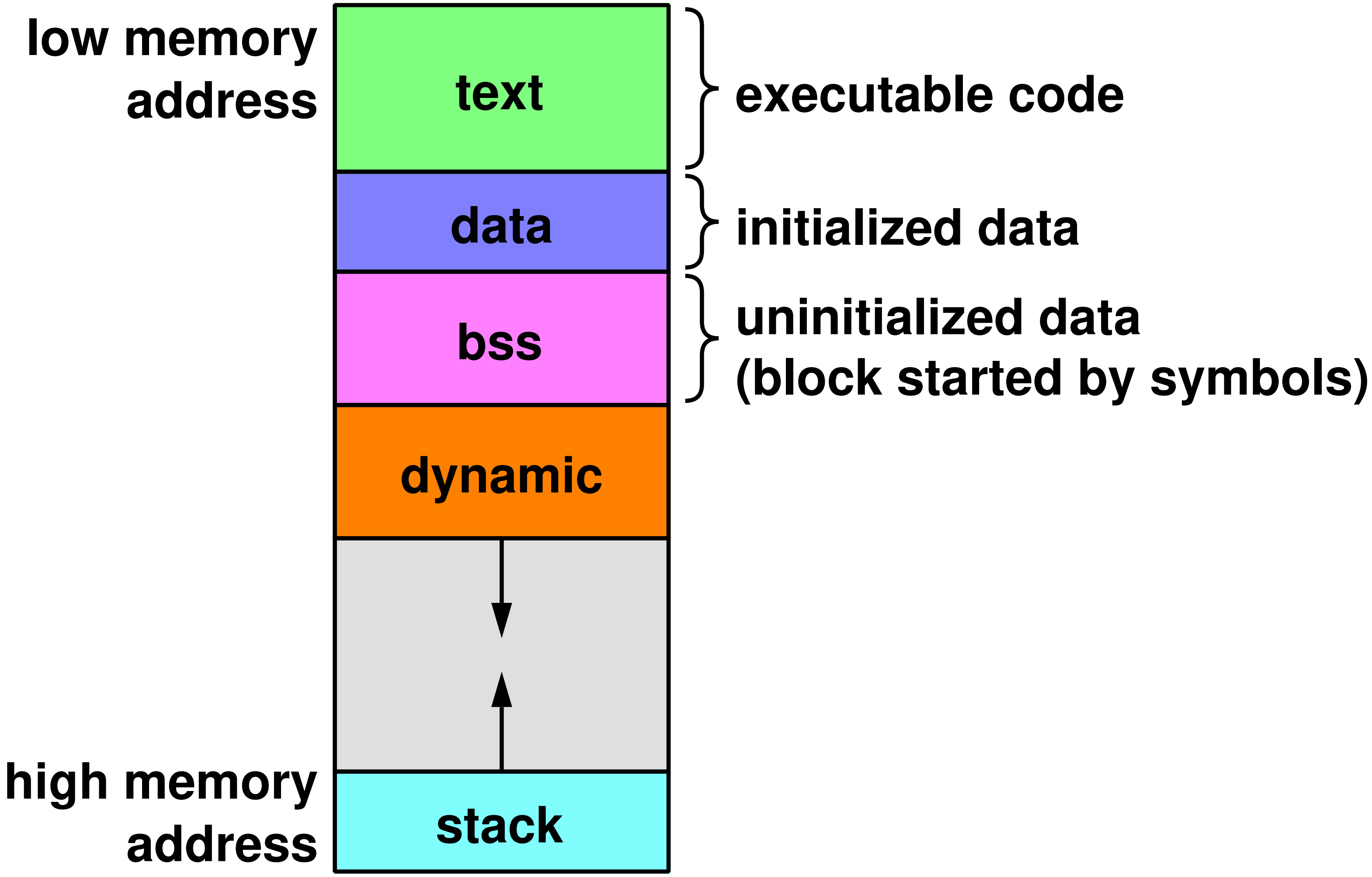
- My color codes for code
  - reserved words at in blue
  - numeric and string constants are in red
  - comments in green
  - black otherwise

# Turing Machine Model of Computation



- A ***Turing Machine*** consists of
  - an infinite tape which is divided into cells, one next to the other *(i.e., **infinite storage**)*
    - one symbol in each cell (or can be a blank symbol)
  - a head that can read and write symbols on the tape and move the tape left and right one (and only one) cell at a time
  - a state register that stores the state of the Turing machine, one of finitely many *(i,e., **finite state**)*
  - a ***finite*** table of instructions that, given the state the machine is currently in and the symbol it is reading on the tape tells the machine to do the following in sequence
    - either erase or write a symbol
    - move the head
    - assume the same or a new state as prescribed

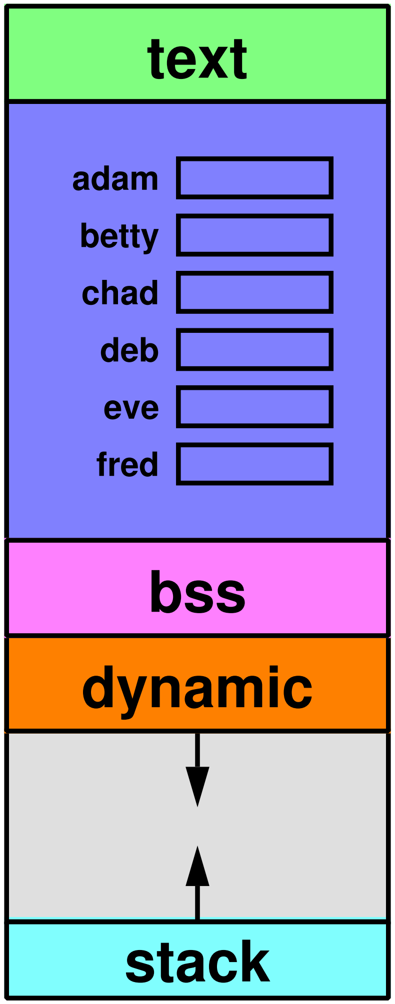
# The Unix Address Space




- This is part of the *tape* of the Turing Machine
  - the rest of the *tape* of the Turing Machine can be reached by using the "*extended address space*"

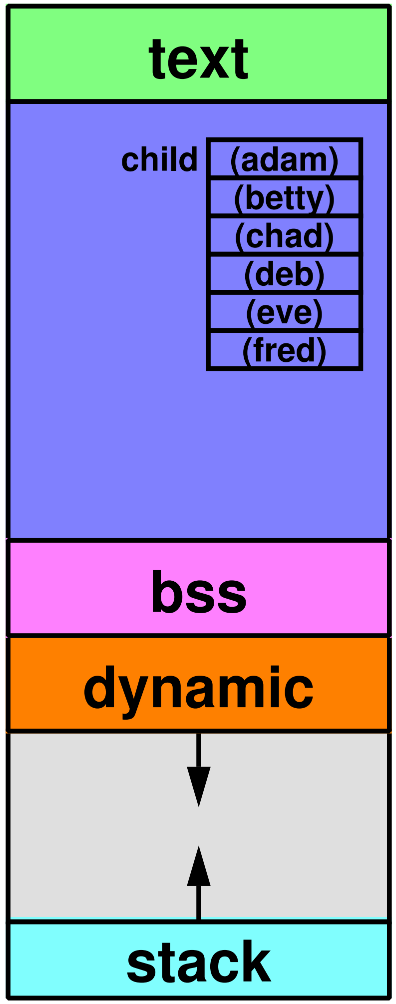
# Note About Naming Objects

- How do you name objects in an address space
  - *"objects"* is the word we use to mean *any* data types (primitive, data structures, pointers)
- Variables
  - name each object
  - a *variable* refers to a *memory location*



# Note About Naming Objects

- How do you name objects in an address space
  - *"objects"* is the word we use to mean *any* data types (primitive, data structures, pointers)
- Variables
  - name each object
  - a *variable* refers to a *memory location*
- Arrays
  - name an object with a *base* and an *index*
- Dynamically create objects do not have names
  - no variable can have a "heap address"
  - need *pointers*



# Note About Naming Objects

- How do you name objects in an address space
  - *"objects"* is the word we use to mean *any* data types (primitive, data structures, pointers)
- Variables
  - name each object
  - a *variable* refers to a *memory location*
- Arrays
  - name an object with a *base* and an *index*
- Dynamically create objects do not have names
  - no variable can have a "heap address"
  - need *pointers*
- For objects that lives in the stack, same name is used for different object instances
  - *function arguments* and *local variables*