Interrupt Handling

- Two objectives
  - handle the interrupt and remove the cause
  - restore what was running before the interrupt
    - kernel may modify saved state on purpose
- Two “actors” in handling the interrupt
  - the hardware goes first
  - the kernel code takes control by running the interrupt handler

Interrupt Handling: HW

- On signal, hardware:
    - Saves state that would be modified by running the interrupt
        - e.g., program counter, registers, mode, etc.
        - where? Depends on the hardware
    - Disables ("masks") device interrupts
        - at least interrupts from the same device
    - Sets supervisor mode (if not set already)
    - Sets PC to first instruction of "signal handler"
        - depends on signal type
        - handlers specified in interrupt vector initialized and loaded at boot time

Interrupt Vector
I/O interrupt handler
System call handler
Page fault handler
…

Interrupt Handling: HW

- To get back, upon executing “return from interrupt” instruction:
  - restores mode
  - restores state saved before the interrupt could run
  - re-enables interrupts

Where's the state of the running process saved?

Where's the state of the running process saved?

PCB

PC
CPU registers
Memory management info
Location of Executable on disk
PID (process identifier)
UID (user executing process)
Scheduling Information
List of open files
Status (running, waiting…)
Saved Kernel SP
Saved User SP
…


On the stack

A Tale of Two Stacks

(it was the best of stacks…)

- Interrupt handler is a program: it needs a stack!
  - so, each process has (at least) two stacks pointers:
    - one when running in user mode
    - a second one when running in kernel mode, to support interrupt handlers
- Why not use the user-level stack?
  - user SP cannot be trusted to be valid or usable
  - user stack may not be large enough, and may spill to overwrite important data
  - security:
    - e.g., kernel could leave sensitive data on stack
      - popping the stack does not erase memory!

Handling Interrupts: SW

- We are now running the interrupt handler!
  - Interrupt handler first pushes the registers' contents (used to run the user process) on the interrupt stack of the currently running process (in the PCB)
    - need registers to run the IH
    - only saves necessary registers (that's why done in SW, not HW)

Registers are typically saved on the interrupt stack, but can be stored anywhere in the PCB

Typical Interrupt Handler Code

```
HandleInterruptX:

  PUSH %Rn        }
     ...          }  only need to save registers not
  PUSH %R1        }  saved by the handler function

  CALL _handleX

  POP %R1         }
     ...          }  restore the registers saved above
  POP %Rn         }

  RETURN_FROM_INTERRUPT
```

Returning from an Interrupt

- Hardware pops saved state of the user process
- Switch to user mode
- Enable interrupts
  - (x86: Depending on content of PSW)
- From exception and system call, may increment PC on return (we don't want to execute again the same instruction!)
  - on exception, handler changes PC at the base of the stack
  - on system call, increment is done by hw when saving user-level state

Can you think of a case when we may want to?

Starting a new process: a recipe

Starting a new process: a recipe

1. Allocate & initialize PCB

Starting a new process:
a recipe

1. Allocate & initialize PCB
2. Setup initial page table (to initialize a new address space)

Starting a new process: a recipe

1. Allocate & initialize PCB
2. Setup initial page table (to initialize a new address space)
3. Load program intro address space

Starting a new process: a recipe

1. Allocate & initialize PCB
2. Setup initial page table (to initialize a new address space)
3. Load program intro address space
4. Allocate user-level and kernel-level stacks.

Starting a new process: a recipe

1. Allocate & initialize PCB
2. Setup initial page table (to initialize a new address space)
3. Load program intro address space
4. Allocate user-level and kernel-level stacks.
5. Copy arguments (if any) to the base/top of user-level stack

Starting a new process: a recipe

1. Allocate & initialize PCB
2. Setup initial page table (to initialize a new address space)
3. Load program intro address space
4. Allocate user-level and kernel-level stacks.
5. Copy arguments (if any) to the base/top of user-level stack
6. Simulate an interrupt

Starting a new process: a recipe

1. Allocate & initialize PCB
2. Setup initial page table (to initialize a new address space)
3. Load program intro address space
4. Allocate user-level and kernel-level stacks.
5. Copy arguments (if any) to the base/top of user-level stack
6. Simulate an interrupt
    a) push on kernel stack initial PC, user SP
    b) [X86] push PSW (supervisor mode off, interrupts enabled)

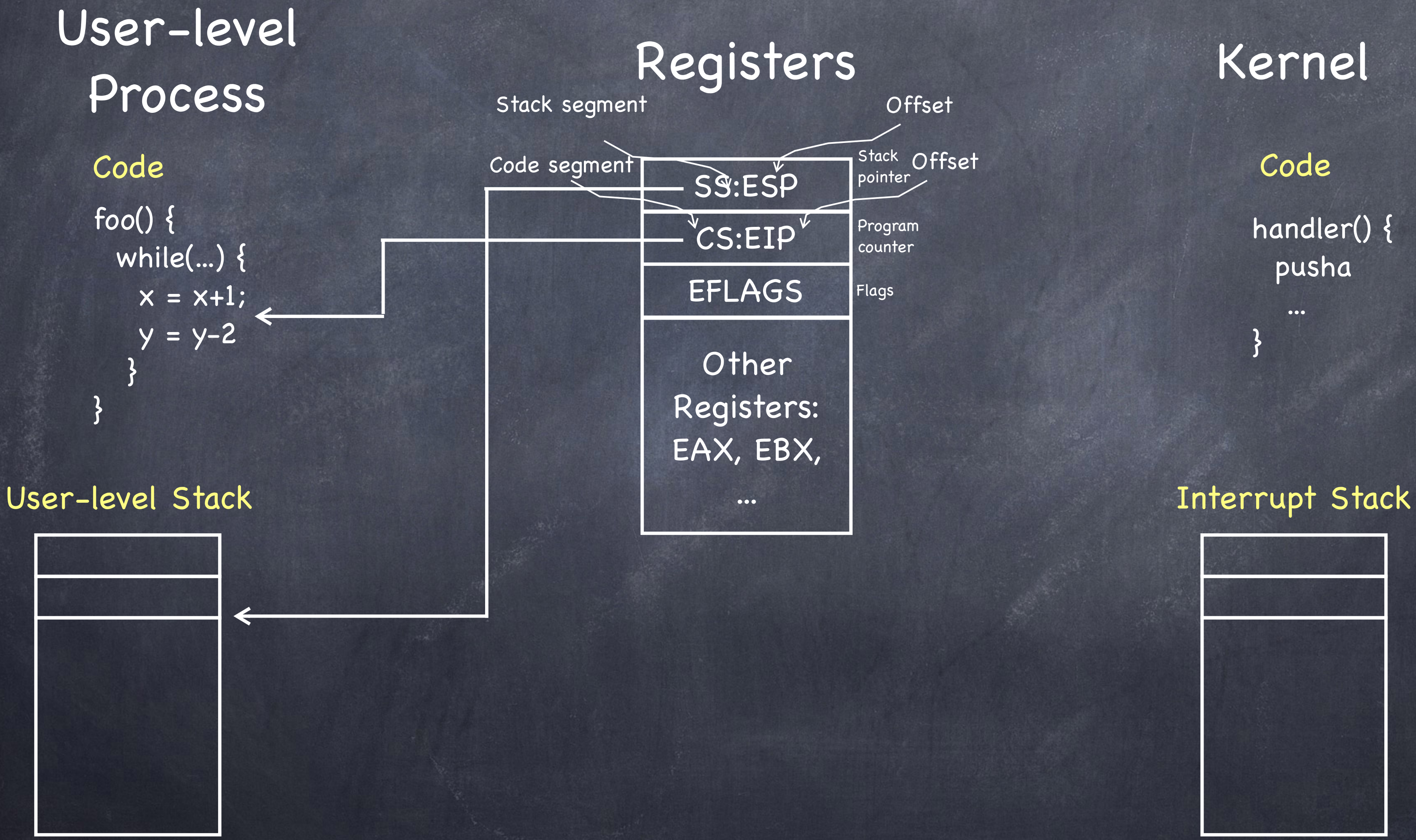
Starting a new process: a recipe

1. Allocate & initialize PCB
2. Setup initial page table (to initialize a new address space)
3. Load program intro address space
4. Allocate user-level and kernel-level stacks.
5. Copy arguments (if any) to the base/top of user-level stack
6. Simulate an interrupt
   a) push on kernel stack initial PC, user SP
   b) [X86] push PSW (supervisor mode off, interrupts enabled)
7. Clear all other registers

Starting a new process: a recipe

1. Allocate & initialize PCB
2. Setup initial page table (to initialize a new address space)
3. Load program intro address space
4. Allocate user-level and kernel-level stacks.
5. Copy arguments (if any) to the base/top of user-level stack
6. Simulate an interrupt
    - a) push on kernel stack initial PC, user SP
    - b) [X86] push PSW (supervisor mode off, interrupts enabled)
7. Clear all other registers
8. RETURN_FROM_INTERRUPT

Interrupt Handling on x86



User-level Process

Code

```
foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}
```

User-level Stack

Registers

Stack segment

Offset

Code segment

SS:ESP

Stack pointer

Offset

CS:EIP

Program counter

EFLAGS

Flags

Other Registers: EAX, EBX, ...

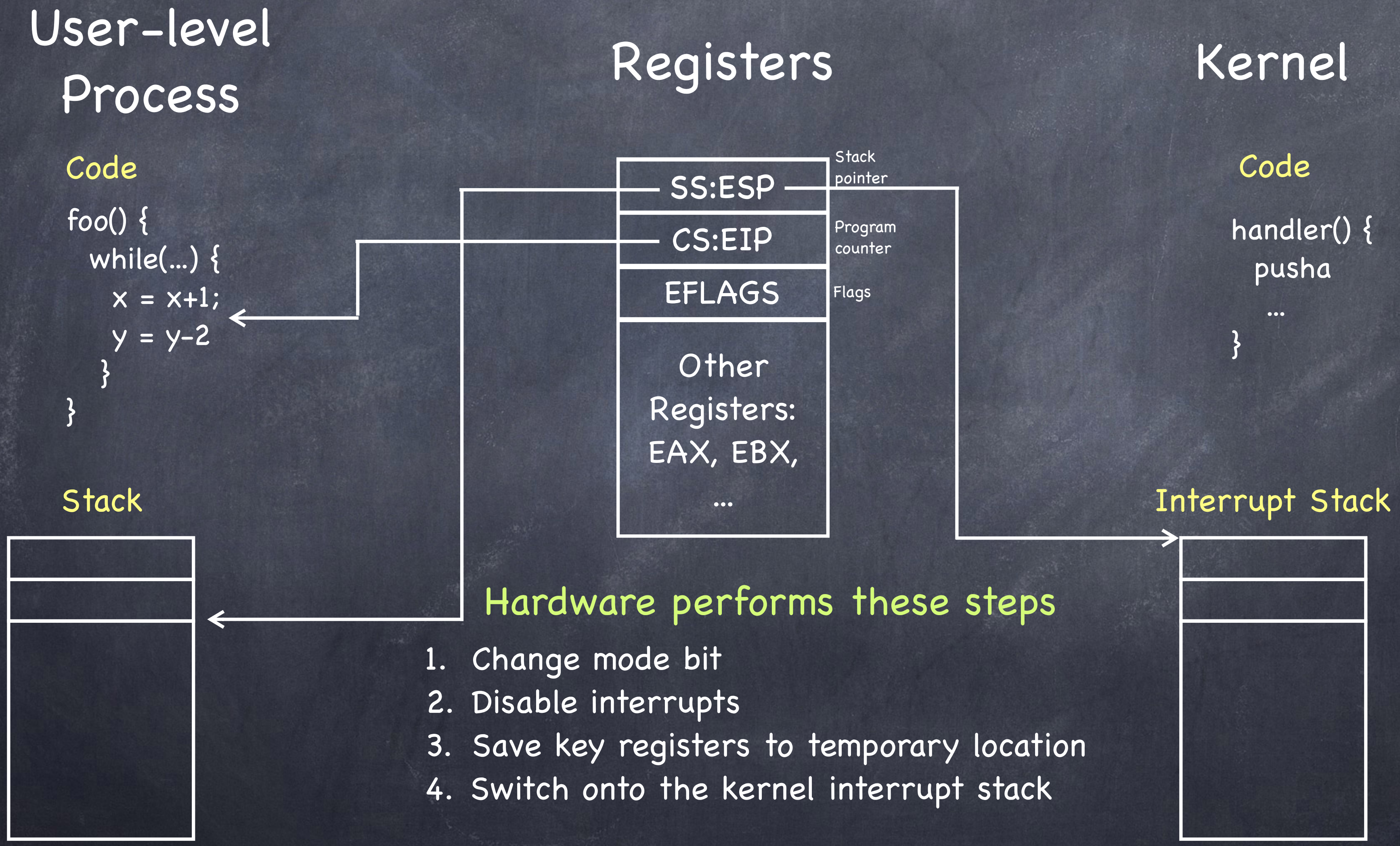
Kernel

Code

```
handler() {
  pusha
  ...
}
```

Interrupt Stack

Interrupt Handling on x86



User-level Process

Code

```
foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}
```

Stack

Registers

SS:ESP — Stack pointer

CS:EIP — Program counter

EFLAGS — Flags

Other Registers: EAX, EBX, ...

Kernel

Code

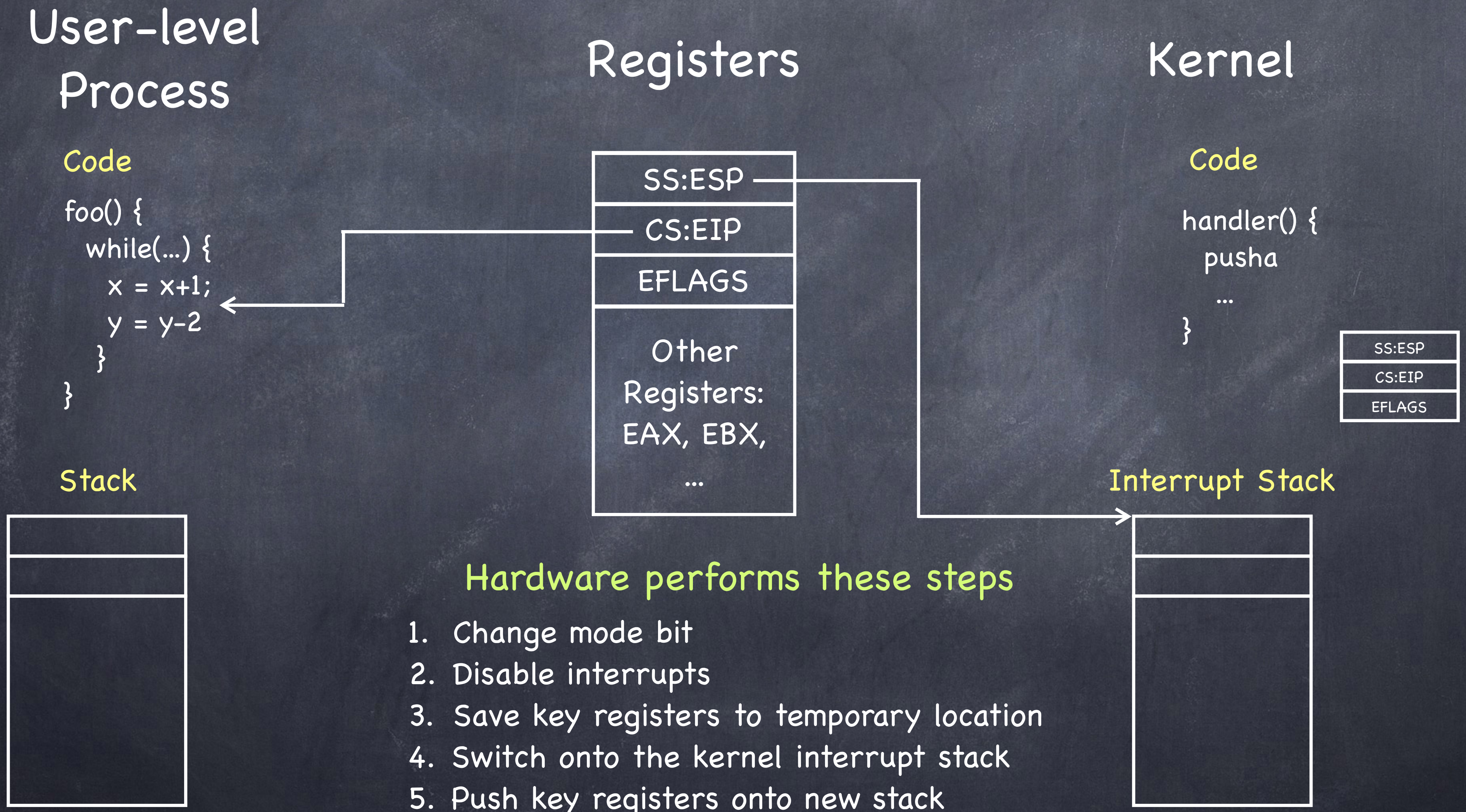
```
handler() {
  pusha
  ...
}
```

Interrupt Stack

Hardware performs these steps

1. Change mode bit
2. Disable interrupts
3. Save key registers to temporary location
4. Switch onto the kernel interrupt stack

Interrupt Handling on x86



User-level Process

Code

```
foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}
```

Stack

Registers

- SS:ESP
- CS:EIP
- EFLAGS
- Other Registers: EAX, EBX, ...

Kernel

Code

```
handler() {
  pusha
  ...
}
```

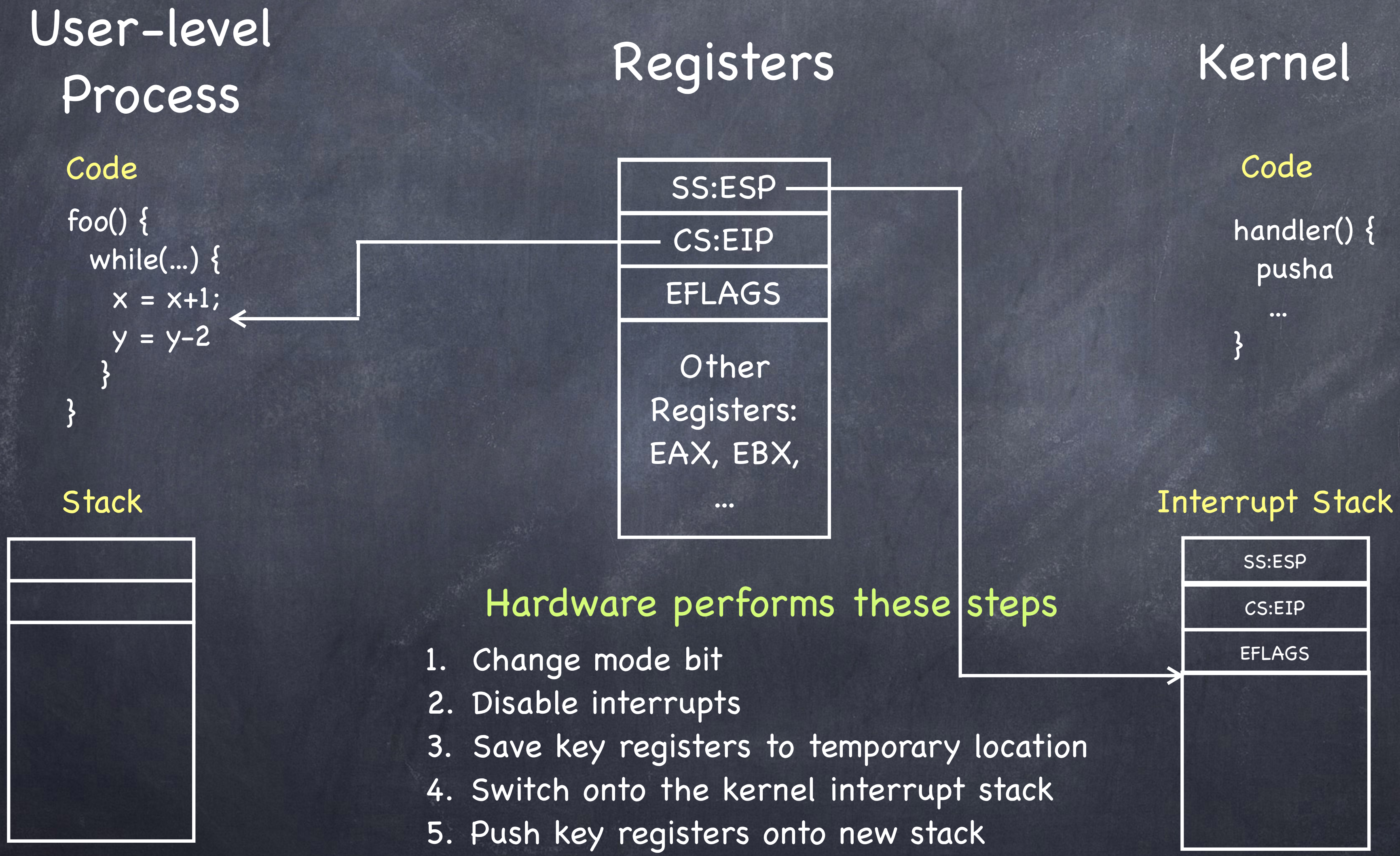
- SS:ESP
- CS:EIP
- EFLAGS

Interrupt Stack

Hardware performs these steps

1. Change mode bit
2. Disable interrupts
3. Save key registers to temporary location
4. Switch onto the kernel interrupt stack
5. Push key registers onto new stack

Interrupt Handling on x86



User-level Process

Code

```
foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}
```

Stack

Registers

SS:ESP

CS:EIP

EFLAGS

Other Registers: EAX, EBX, ...

Kernel

Code

```
handler() {
  pusha
  ...
}
```

Interrupt Stack

SS:ESP

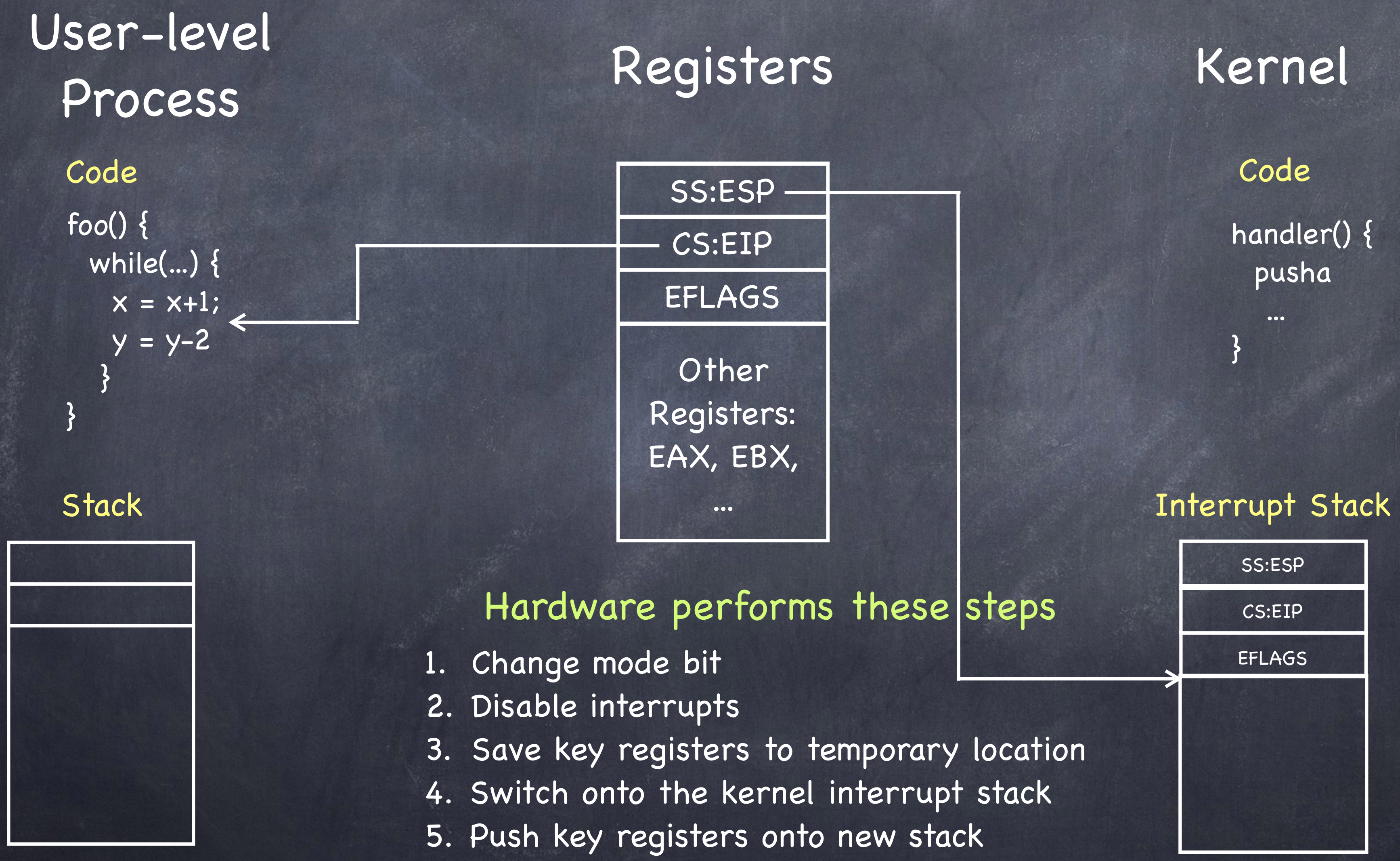
CS:EIP

EFLAGS

Hardware performs these steps

1. Change mode bit
2. Disable interrupts
3. Save key registers to temporary location
4. Switch onto the kernel interrupt stack
5. Push key registers onto new stack

Interrupt Handling on x86



User-level Process

Code

```
foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}
```

Stack

Registers

Kernel

Code

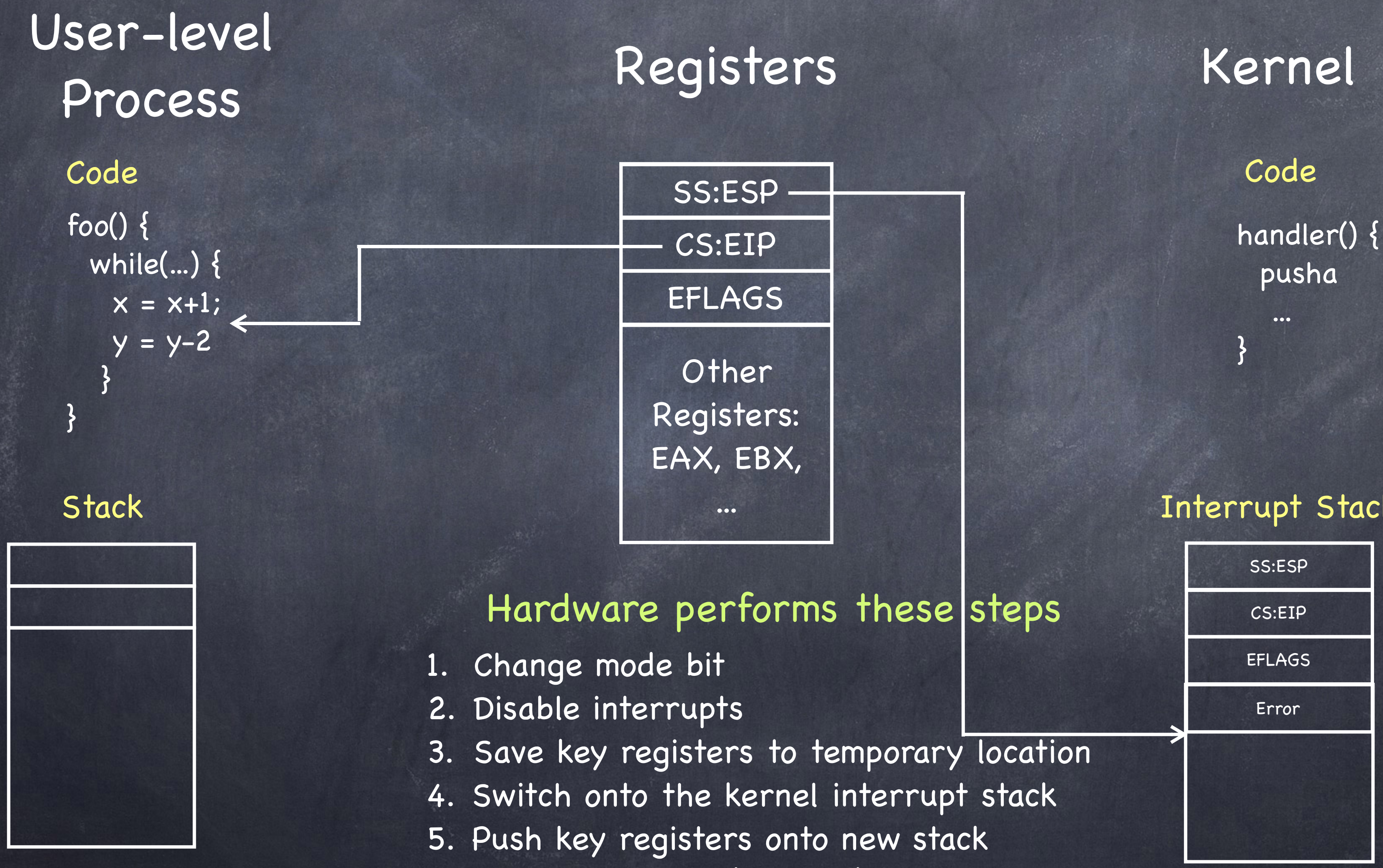
```
handler() {
  pusha
  ...
}
```

Interrupt Stack

Hardware performs these steps

1. Change mode bit
2. Disable interrupts
3. Save key registers to temporary location
4. Switch onto the kernel interrupt stack
5. Push key registers onto new stack
6. Save error code (optional)

Interrupt Handling on x86



User-level Process

Code

```
foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}
```

Stack

Registers

SS:ESP

CS:EIP

EFLAGS

Other Registers: EAX, EBX, ...

Kernel

Code

```
handler() {
  pusha
  ...
}
```

Interrupt Stack

SS:ESP

CS:EIP

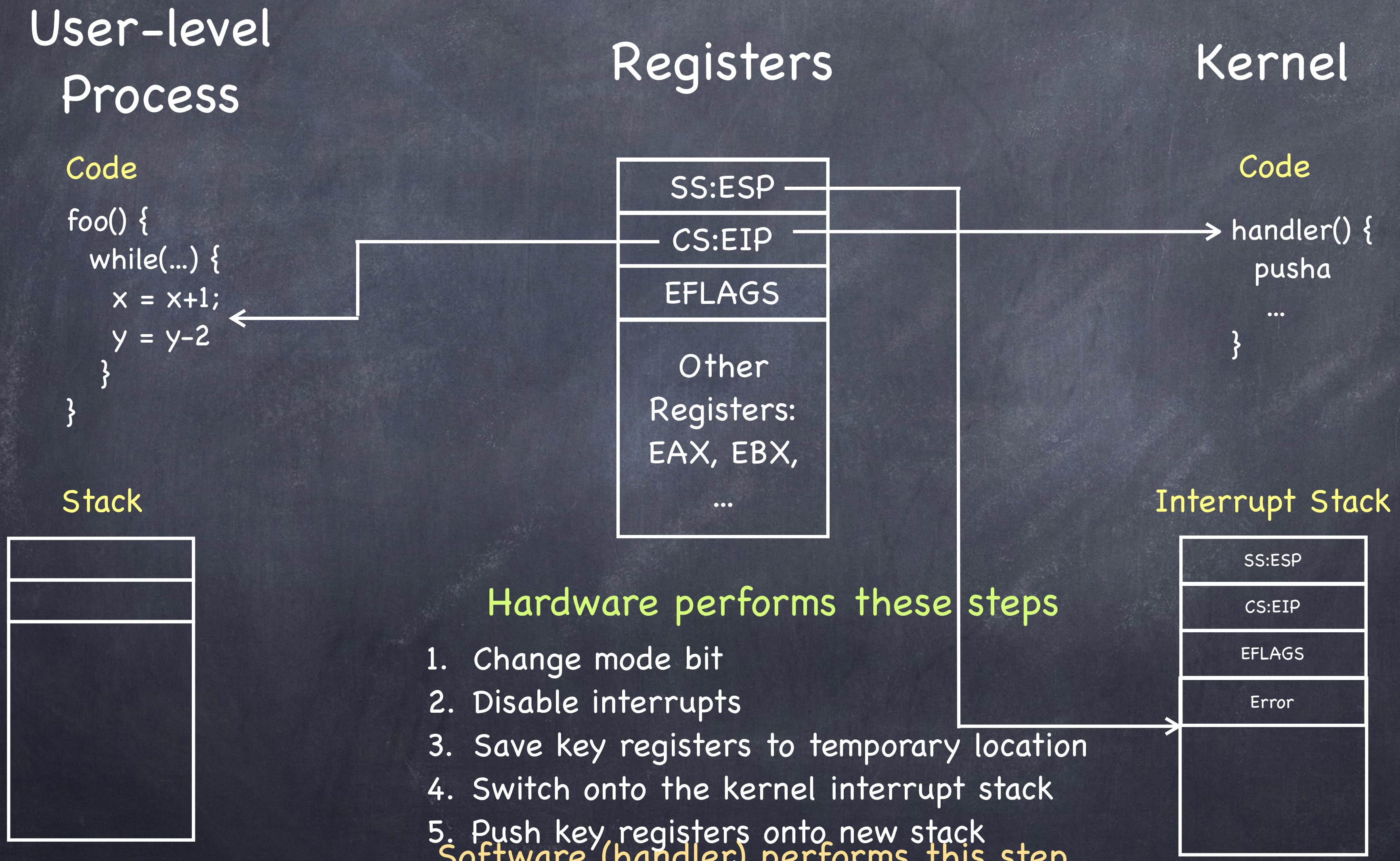
EFLAGS

Error

Hardware performs these steps

1. Change mode bit
2. Disable interrupts
3. Save key registers to temporary location
4. Switch onto the kernel interrupt stack
5. Push key registers onto new stack
6. Save error code (optional)

Interrupt Handling on x86



User-level Process

Code

```
foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}
```

Stack

Registers

Kernel

Code

```
handler() {
  pusha
  ...
}
```

Interrupt Stack

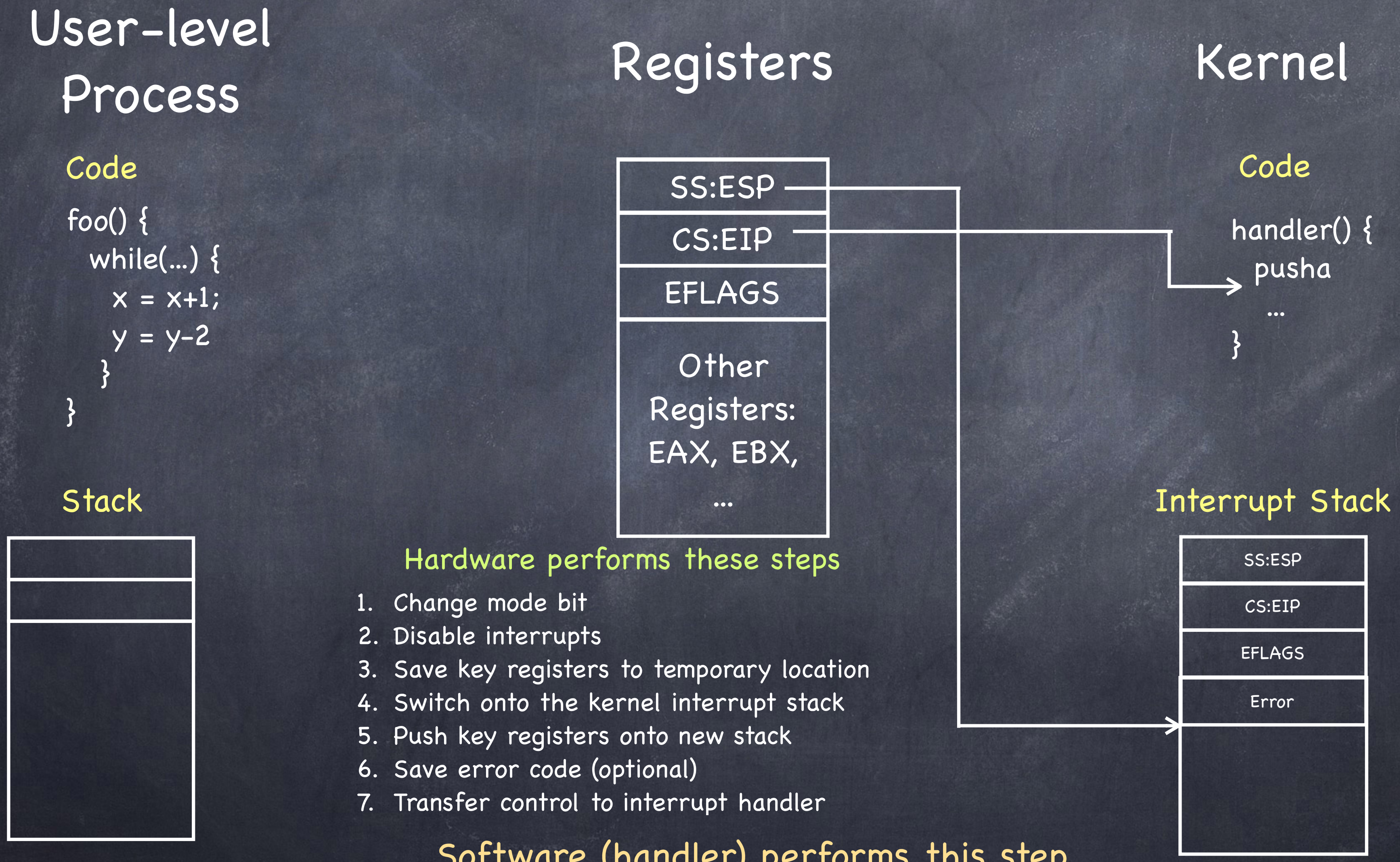
Hardware performs these steps

1. Change mode bit
2. Disable interrupts
3. Save key registers to temporary location
4. Switch onto the kernel interrupt stack
5. Push key registers onto new stack
6. Save error code (optional)
7. Transfer control to interrupt handler

Software (handler) performs this step

8. Handler pushes select registers on stack

Interrupt Handling on x86



User-level Process

Code

```
foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}
```

Stack

Registers

Kernel

Code

```
handler() {
  pusha
  ...
}
```

Interrupt Stack

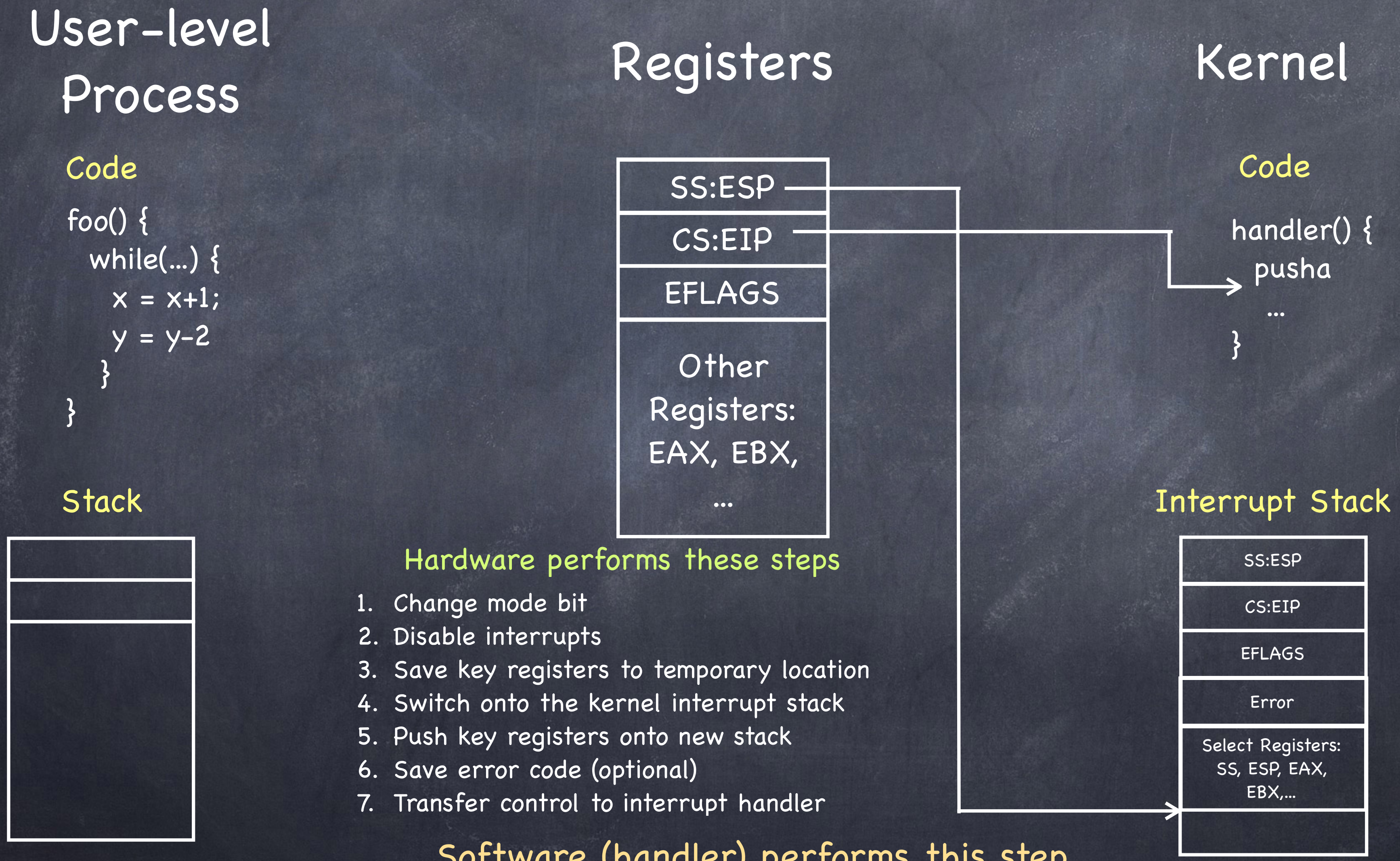
Hardware performs these steps

1. Change mode bit
2. Disable interrupts
3. Save key registers to temporary location
4. Switch onto the kernel interrupt stack
5. Push key registers onto new stack
6. Save error code (optional)
7. Transfer control to interrupt handler

Software (handler) performs this step

8. Handler pushes select registers on stack

Interrupt Handling on x86



User-level Process

Code

```
foo() {
  while(...) {
    x = x+1;
    y = y-2
  }
}
```

Stack

Registers

SS:ESP

CS:EIP

EFLAGS

Other Registers: EAX, EBX, ...

Kernel

Code

```
handler() {
  pusha
  ...
}
```

Interrupt Stack

SS:ESP

CS:EIP

EFLAGS

Error

Select Registers: SS, ESP, EAX, EBX,...

Hardware performs these steps

1. Change mode bit
2. Disable interrupts
3. Save key registers to temporary location
4. Switch onto the kernel interrupt stack
5. Push key registers onto new stack
6. Save error code (optional)
7. Transfer control to interrupt handler

Software (handler) performs this step

8. Handler pushes select registers on stack

Interrupt Safety

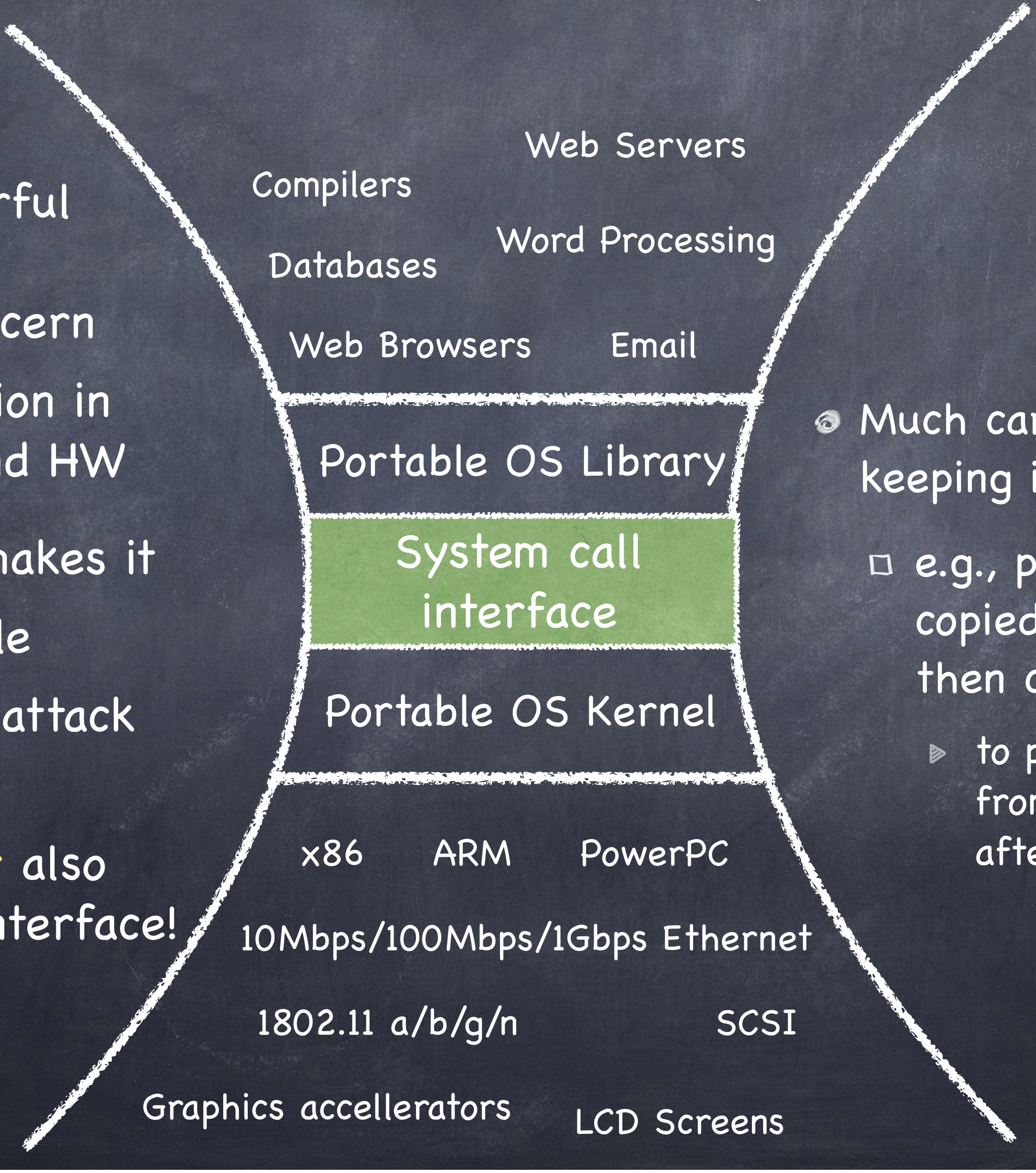
- Kernel should disable device interrupts as little as possible
    - interrupts are best serviced quickly
- Thus, device interrupts are often disabled selectively
    - e.g., clock interrupts enabled during disk interrupt handling
- This leads to potential “race conditions”
    - system’s behavior depends on timing of asynchronous (and thus uncontrollable) events

System calls

- Programming interface to the services the OS provides:
  - read input/write to screen
  - create/read/write/delete files
  - create new processes
  - send/receive network packets
  - get the time / set alarms
  - terminate current process
  - …

The Skinny

- Simple and powerful interface allows separation of concern
  - Eases innovation in user space and HW
- "Narrow waist" makes it
  - highly portable
  - robust (small attack surface)
- Internet IP layer also offers a skinny interface!



Web Servers

Compilers

Word Processing

Databases

Web Browsers

Email

Portable OS Library

System call interface

Portable OS Kernel

x86

ARM

PowerPC

10Mbps/100Mbps/1Gbps Ethernet

1802.11 a/b/g/n

SCSI

Graphics accellerators

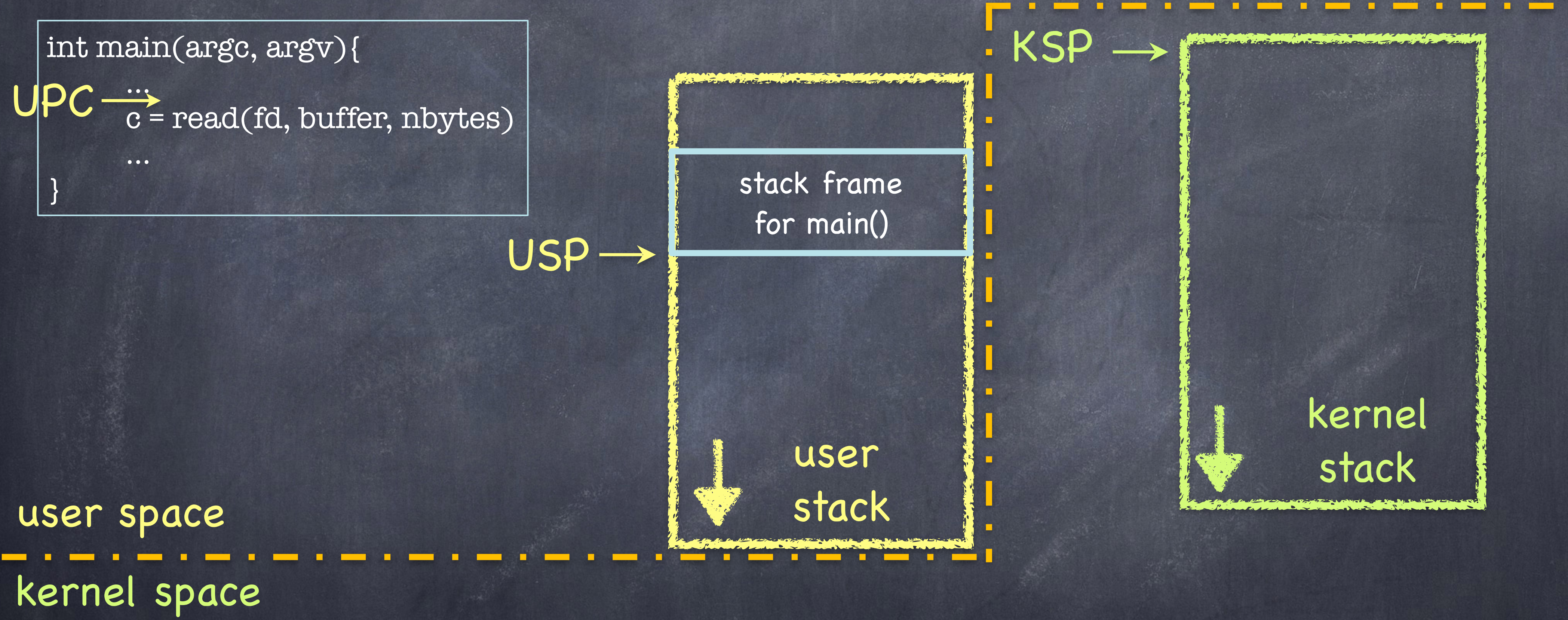
LCD Screens

- Much care spent in keeping interface secure
  - e.g., parameters first copied to kernel space, then checked
    - to prevent user program from changing them after they are checked!

Executing a System Call

- Process:
  - Calls system call function in library
  - Places arguments in registers and/or pushes them onto user stack
  - Places syscall type in a dedicated register
  - Executes `syscall` machine instruction
- Kernel
  - Executes syscall interrupt handler
  - Places result in dedicated register
  - Executes RETURN_FROM_INTERRUPT
- Process:
  - Executes RETURN_FROM_FUNCTION

Executing read System Call



```
int main(argc, argv){
        ...
    c = read(fd, buffer, nbytes)
        ...
}
```

UPC: user program counter

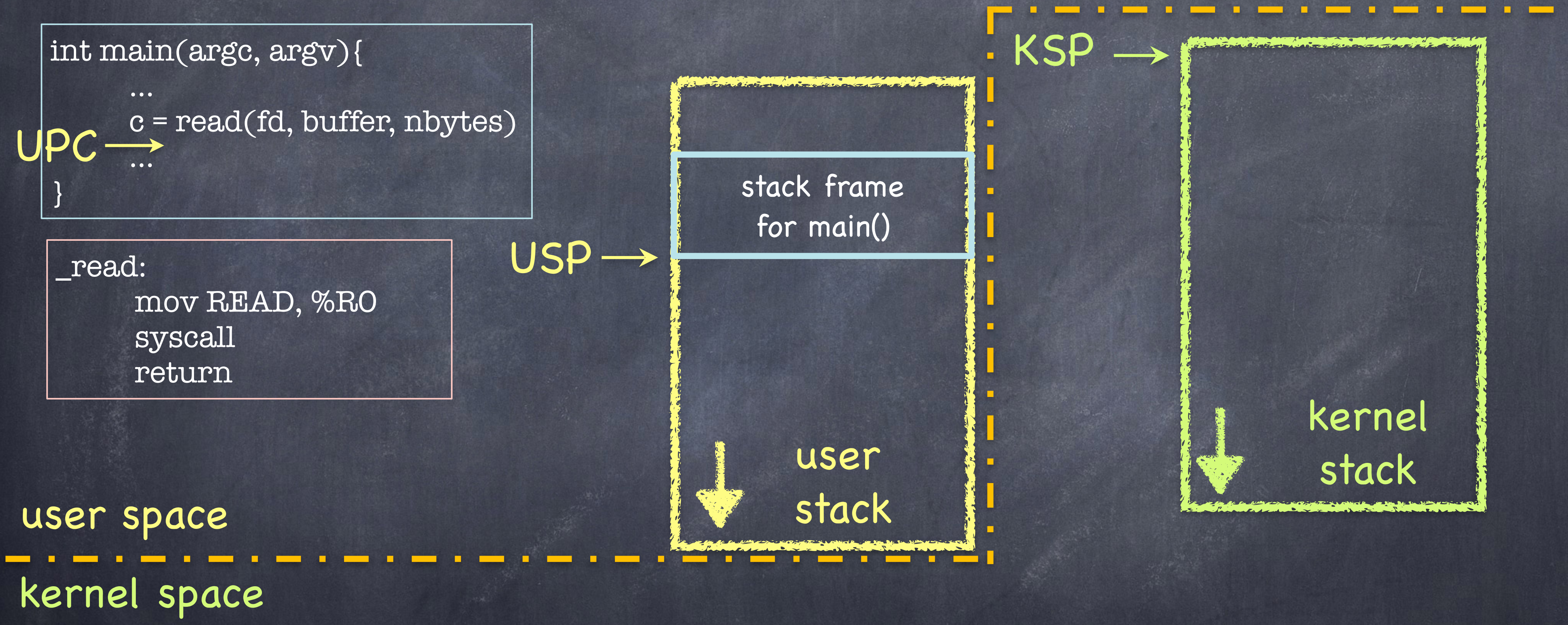
USP: user stack pointer

KSP: kernel stack pointer

KPC: kernel program counter

note: kernel stack is empty while user process running

Executing read System Call



```
int main(argc, argv){
        ...
        c = read(fd, buffer, nbytes)
        ...
}
```

```
_read:
        mov READ, %R0
        syscall
        return
```

UPC: user program counter

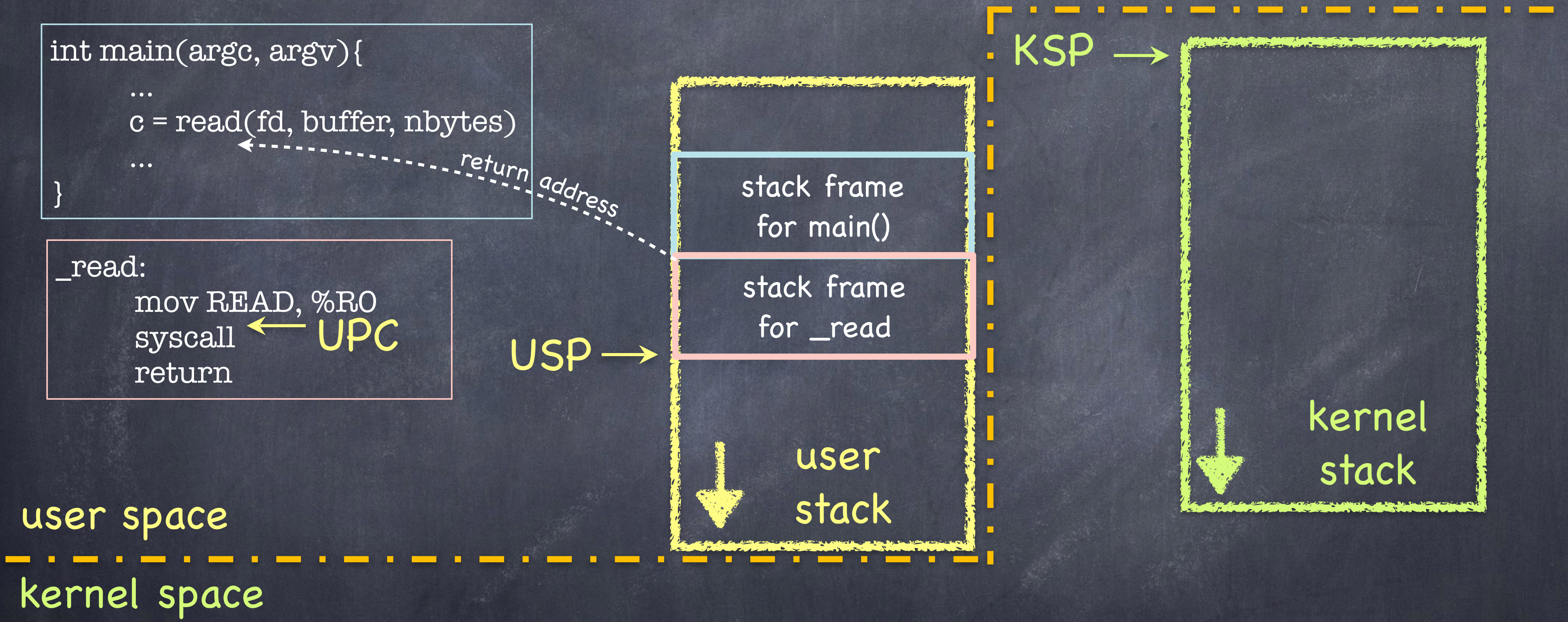
USP: user stack pointer

KSP: kernel stack pointer

KPC: kernel program counter

note: kernel stack is empty while user process running

Executing read System Call



```
int main(argc, argv){
        ...
        c = read(fd, buffer, nbytes)
        ...
}
```

```
_read:
        mov READ, %R0
        syscall   ← UPC
        return
```

return address

USP →

stack frame for main()

stack frame for _read

user stack

KSP →

kernel stack

user space

kernel space

UPC: user program counter

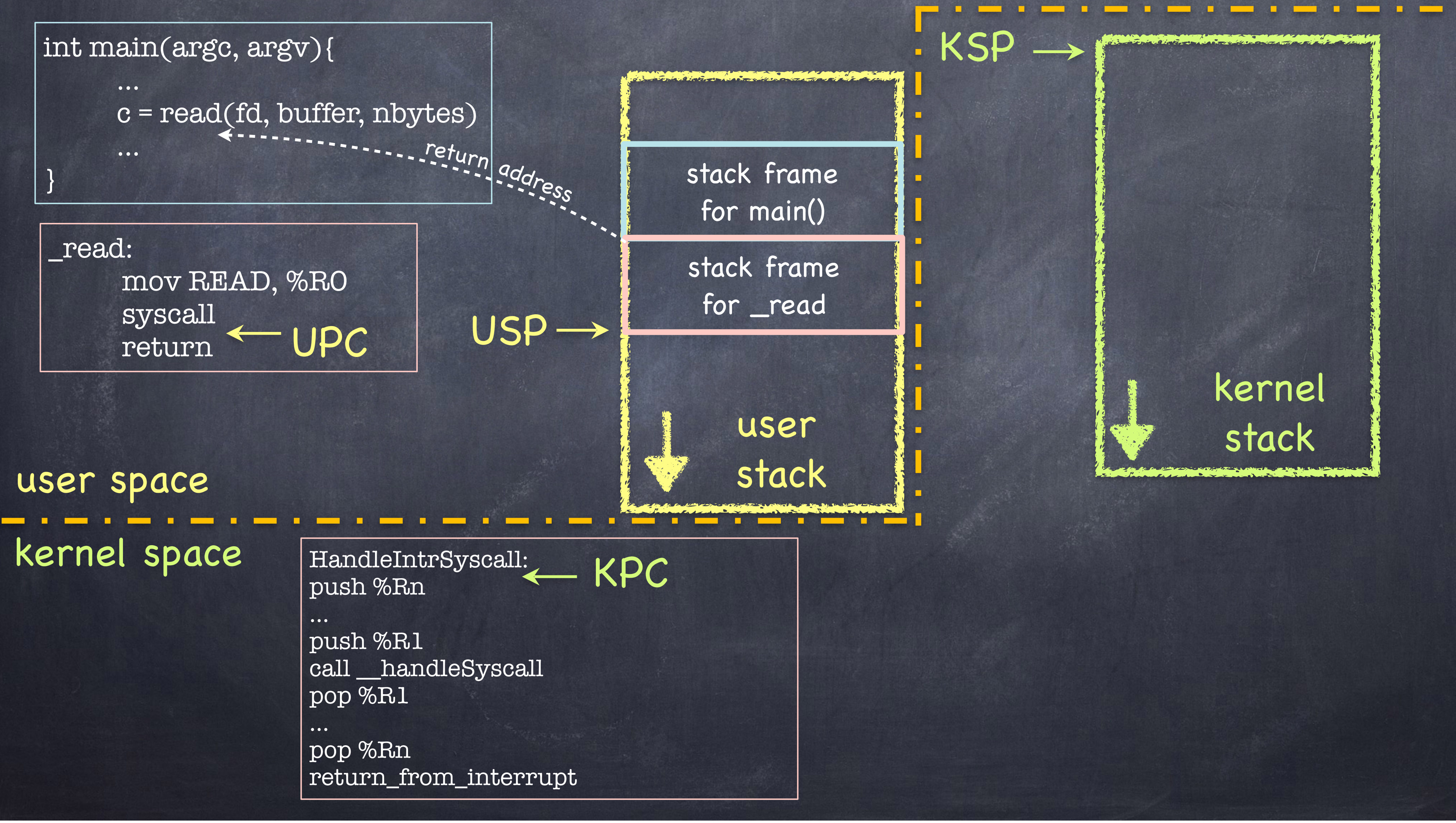
USP: user stack pointer

KSP: kernel stack pointer

KPC: kernel program counter

note: kernel stack is empty while user process running

Executing read System Call

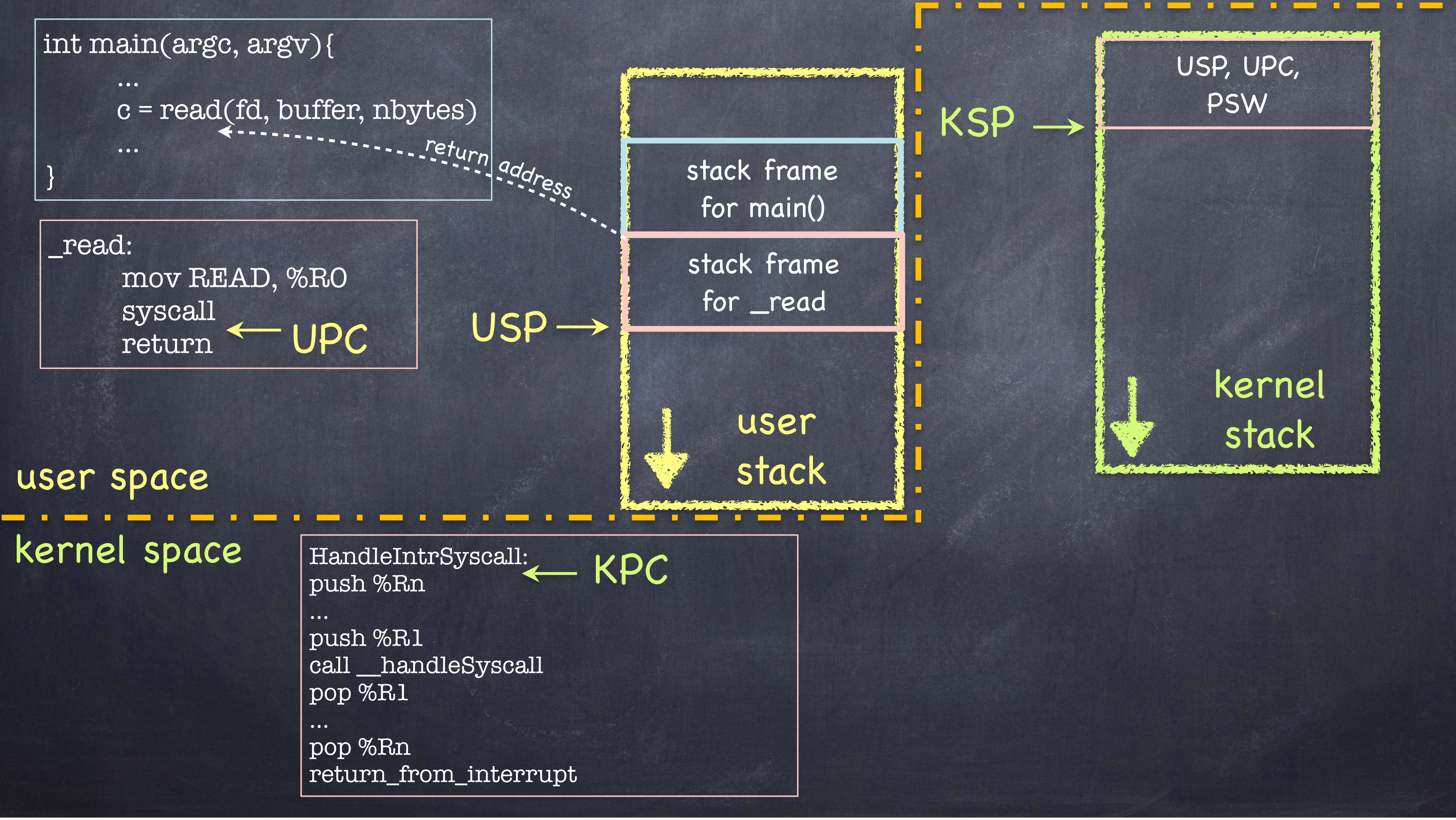


```
int main(argc, argv){
        ...
        c = read(fd, buffer, nbytes)
        ...
}
```

```
_read:
        mov READ, %R0
        syscall
        return
```

```
HandleIntrSyscall:
push %Rn
...
push %R1
call __handleSyscall
pop %R1
...
pop %Rn
return_from_interrupt
```

Executing read System Call

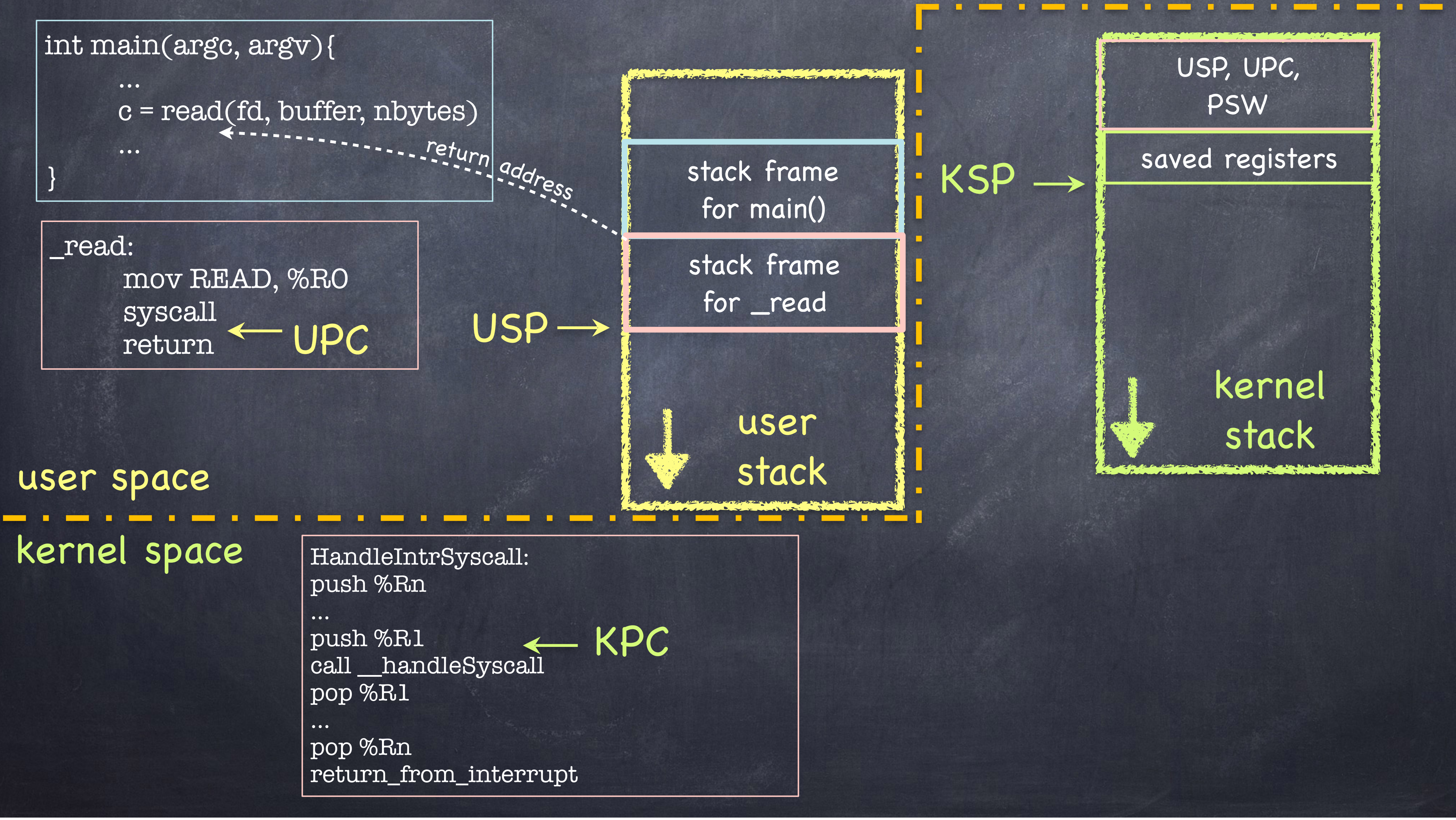


```
int main(argc, argv){
        ...
        c = read(fd, buffer, nbytes)
        ...
}
```

```
_read:
        mov READ, %R0
        syscall
        return
```

```
HandleIntrSyscall:
push %Rn
...
push %R1
call __handleSyscall
pop %R1
...
pop %Rn
return_from_interrupt
```

Executing read System Call



```
int main(argc, argv){
        ...
        c = read(fd, buffer, nbytes)
        ...
}
```

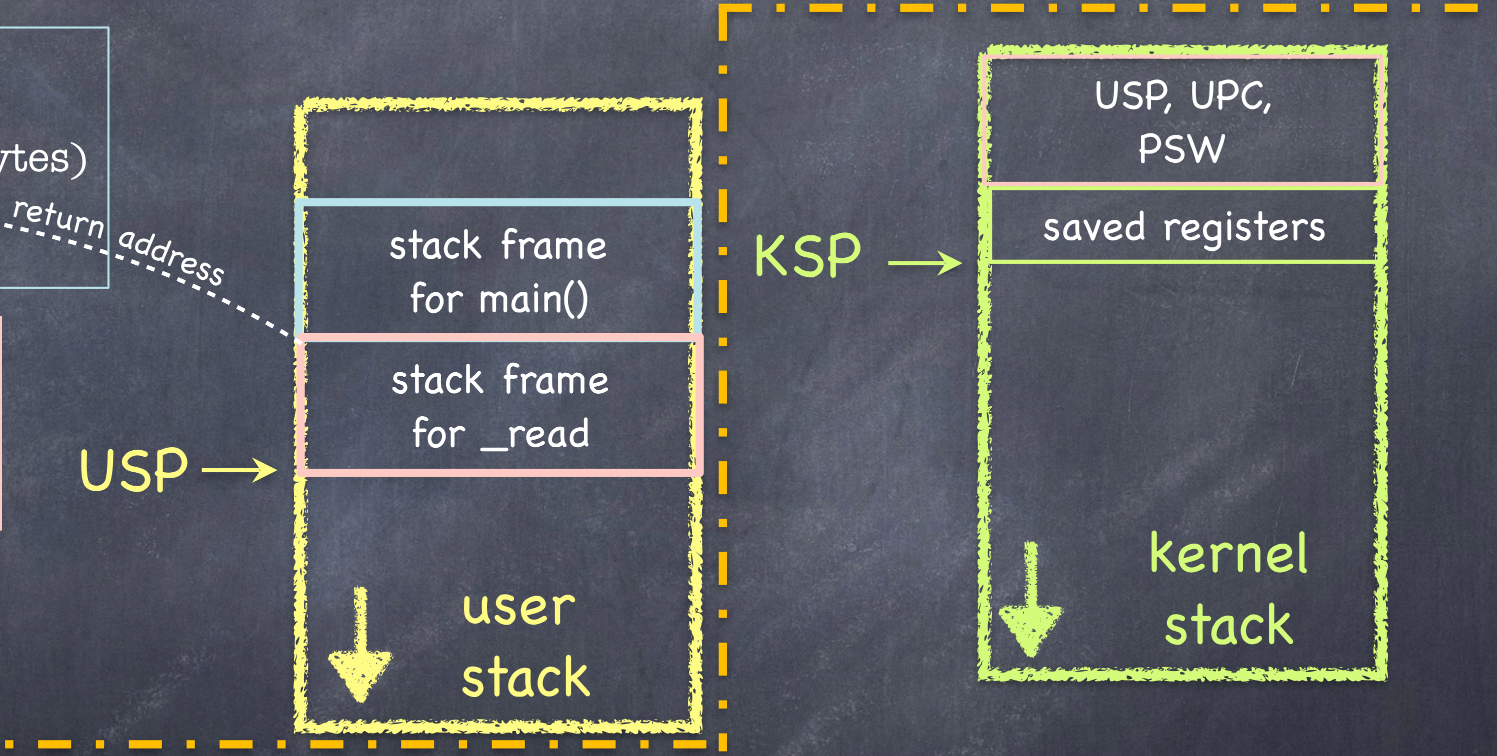
```
_read:
        mov READ, %R0
        syscall
        return  ← UPC
```

```
HandleIntrSyscall:
push %Rn
...
push %R1        ← KPC
call __handleSyscall
pop %R1
...
pop %Rn
return_from_interrupt
```

Executing read System Call

```
int main(argc, argv){
        ...
        c = read(fd, buffer, nbytes)
        ...
}
```

```
_read:
        mov READ, %R0
        syscall
        return   ← UPC
```



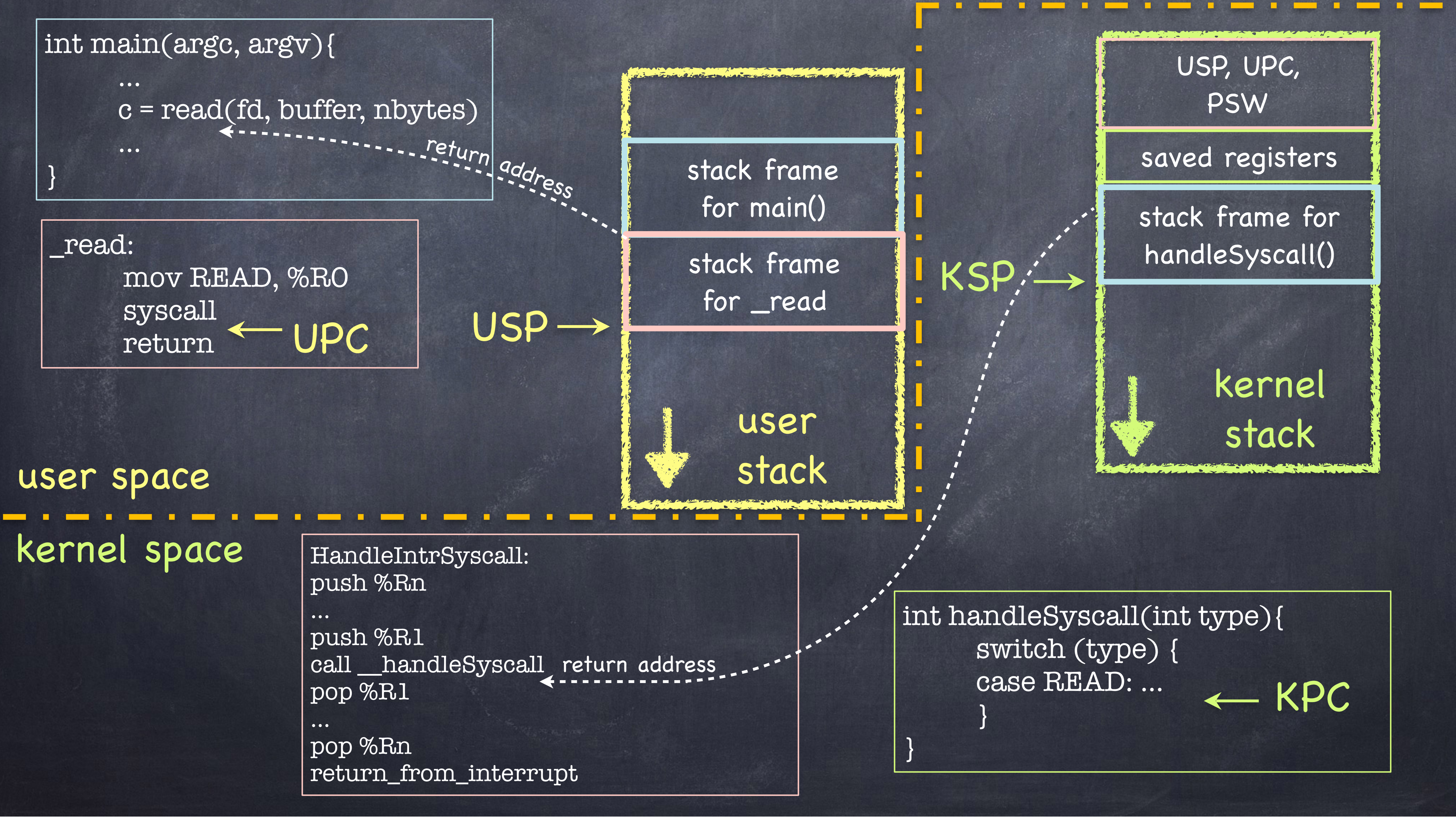
user space

kernel space

```
HandleIntrSyscall:
push %Rn
...
push %R1
call __handleSyscall  ← KPC
pop %R1
...
pop %Rn
return_from_interrupt
```

```
int handleSyscall(int type){
        switch (type) {
        case READ: ...
        }
}
```

Executing read System Call



```
int main(argc, argv){
        ...
        c = read(fd, buffer, nbytes)
        ...
}
```

return address

```
_read:
        mov READ, %R0
        syscall
        return
```

UPC

USP

stack frame for main()

stack frame for _read

user stack

user space

kernel space

KSP

USP, UPC, PSW

saved registers

stack frame for handleSyscall()

kernel stack

```
HandleIntrSyscall:
push %Rn
...
push %R1
call __handleSyscall
pop %R1
...
pop %Rn
return_from_interrupt
```

return address

```
int handleSyscall(int type){
        switch (type) {
        case READ: ...
        }
}
```

KPC

What if read needs to block?

- read may need to block if
    - It reads from a terminal
    - It reads from disk, and block is not in cache
    - It reads from a remote file server

We should run another process!