CS4410

Operating Systems

Lecture 5:
Abstractions III: IPC
Interrupts and Signals
Life cycle of a process

Rachit Agarwal

Context for today's lecture

- By the end of today's lecture, you will know most of the basic terminology in OS
    - Virtual cores (threads) and processes
        - Thread/Process control blocks, stacks, PC, SP, …
    - Address space, and translation
    - Files, pipes and sockets (today)
    - Interrupts, system calls, signals (today)

- And you already understand some basic ideas in protection/isolation/sharing
    - Dual mode operations

- You will be ready for a deep dive into OS mechanisms:
    - Synchronization and scheduling
    - Memory management (including address translation)
    - Storage stack
    - Network stack

Goal of Today's Lecture

- Revisit "Everything is a file" idea
    - Third set of abstractions: Files, Pipes and Sockets
- Understand interrupts and signals
- Understand the life cycle of a process

Abstraction III: I/O

Recap: Everything is a “File”

- **A radical idea**
  - Proposed by Dennis Ritchie and Ken Thompson in 1974
  - In their seminal paper on UNIX called “The UNIX Time-Sharing System”

- **Core idea: we should have identical interfaces for:**
  - Files on disk
  - Networking (sockets)
  - Devices (terminals, printers, etc.)
  - Local interprocess communication (pipes, sockets)

File

- **Named collection of data**
- Has a **name:** readable by humans
- Has a **file descriptor**:
    - unique identifier (handle) used by a process to identify an opened file
- May have **data:** Text, binary, serialized objects, …
- Has associated **metadata**: size, modification time, access control, etc.
- Can execute system calls to perform operations:
    - **open()**, **read()**, **write()**, and **close()**
    - And many others …
- Kernel ensures protection

Key Design Ideas

- **Uniformity:** everything is a file
- **open() before use:** Provides opportunity for access control and arbitration
- **Byte-oriented:** Least common denominator
    - OS hides underlying details:
        - Block-based data transfers? Sure.
        - Stream data transfers? Sure.
- **Kernel buffered read() and write()**
    - Helpful to make everything byte-oriented
    - Process is **blocked** while waiting for return
    - Complete in background
        - Writes return immediately
    - Enables a “global” buffer management (eg., taking caches into account)
- **Explicit close()**

Interprocess Communication

- **What if two processes wish to communication with one another?**
    - What are the possible options?

- **One option: shared memory address space**
    - Processes read/write to this shared address space w/o kernel intervention
    - Potential protection violation

- **Another option: use a file (on slower storage)**
    - Producer process writes to a file; consumer process reads.
    - Kernel mediates read/write operations
        - Protection guaranteed by the kernel
    - Problem?
        - High overheads

- Other options: IPC and Sockets

“Pipes” for Interprocess Communication

- **Create an *in-memory* queue**
    - Data written by producer process is written to the queue
    - Consumer processes can read from the queue
    - Use a file interface to enable reads and writes
        - Use two file descriptors: one for each of read and write
    - Ask the kernel to execute operations via syscalls.

- **Questions**
    - What if A generates data faster than B can consume it?
    - What if B consumes data faster than A generates it?

- **Solution:**
    - It is okay—file read and write operations are blocking.

- **This queue is called a “pipe”**

"Sockets" for *Remote* Interprocess Communication

- **What if the two processes are running on two different physical servers?**
    - With a network sitting in the middle?
    - What could we do?

- **Sockets!**
    - Similar to pipe, is a file
- Create an in-memory queue/file at each process
    - Sending process writes to its queue (via kernel mediation)
    - Kernel transfers data to the other queue
    - Reading process reads from its queue
- OS enables correctness: ensures the two queues have the same "view"
    - The same data and the same ordering
    - Using a reliable, in-order, data delivery protocol over the network

Interrupts and Signals

Interrupts

A **response by the processor** to an **event** that needs attention from **software**

- Some **events** trigger an **interrupt condition** that alerts the processor
  - "When permitted, please interrupt the currently executing code"
  - So that the **event** can be processed in a timely manner

- Upon receiving an interrupt, **response by the processor**
  - Suspends its current activities, whenever safe
  - Saves its state
  - Executes a function called **interrupt handler**
  - Resumes operations once interrupt handler finishes

- Interrupts can be:
  - **Maskable**: Can by turned off by the CPU for critical processing
  - **Non-maskable**: Indicate serious errors
    - E.g., power out warning, unrecoverable memory error, etc.

Type of interrupts (slightly misusing the terminology)

Exceptions

- "Software interrupts"
- process missteps (e.g. division by zero)
- attempt to perform a privileged instruction
  - sometime on purpose! (breakpoints)
- synchronous/non-maskable

System calls/traps

- requests OS service
- synchronous/non-maskable

Hardware Interrupts

- HW device requires OS service
  - timer, I/O device, interprocessor
- asynchronous/maskable

Interrupt handling

- Two objectives
    - Handle the interrupt and remove the cause
    - Restore whatever was running before the interrupt
        - Saved state may have been modified on purpose

- Two “actors” in handling the interrupts
    - The hardware goes first
    - The kernel code takes control by running the **interrupt handler**

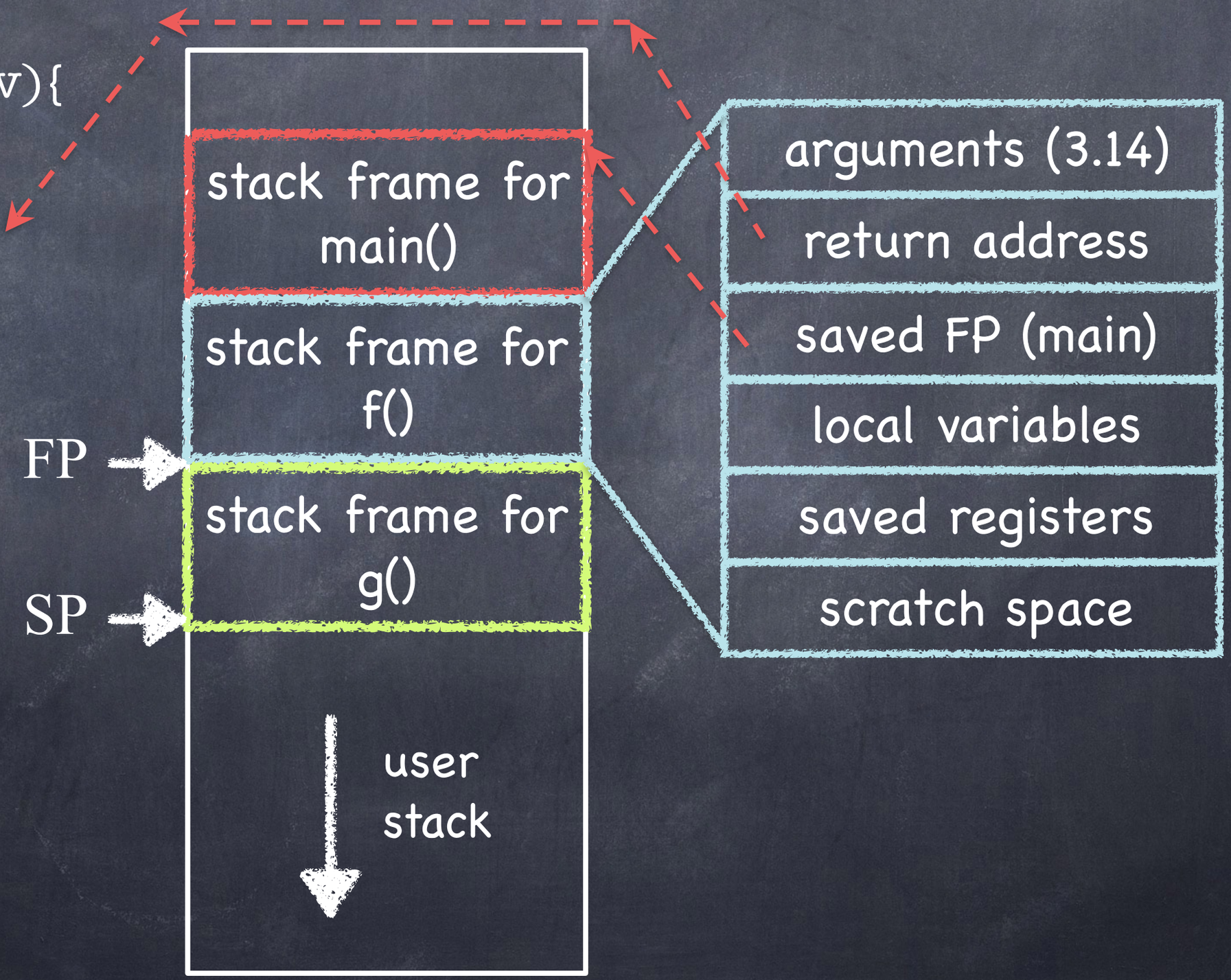
Review: stack (aka call stack)

```
int main(argc, argv){
      …
      f(3.14)
      …
}

int f(x){
      …
      g();
      …
}

int g(y){
      …
}
```

PC



A tale of two Stack Pointers

- Interrupt Handler is a block of code: it needs a stack!
    - So, each process has two stack pointers (SP)
        - One when running in the user mode
        - A second one when running in the kernel mode

- Why not use the user-level SP?
    - User SP cannot be trusted

Handling interrupts

- **Hardware:** Upon an interrupt, the hardware
    - Sets supervisor mode (if not already set)
    - Disable (masks) interrupts
    - Pushes PC, SP, etc. on interrupt stack
    - Sets PC to point to the first instruction of the appropriate interrupt handler
        - Depends on interrupt type
        - Specified in an interrupt vector (loaded at boot time)

- **Software:** We are now running the interrupt handler
    - Pushes the registers' contents (for user process) to the interrupt stack
        - Need registers to run the interrupt handler
        - Only saves necessary registers
        - That is why done in software, not hardware

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
```

Interrupt Handling on x86

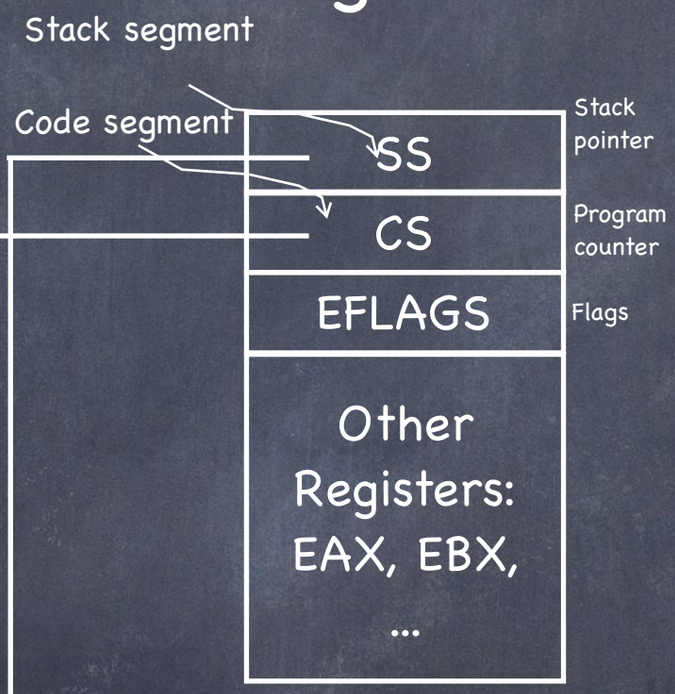
User-level Process

Code

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

Stack

Registers



Kernel

Code

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

Interrupt Stack

Interrupt Handling on x86

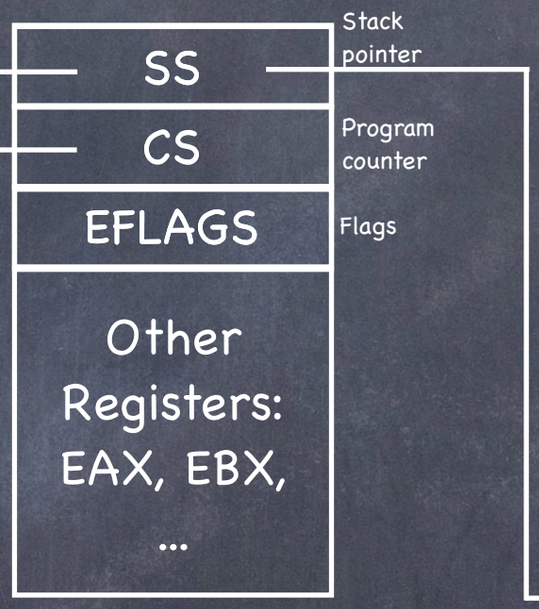
User-level Process

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

Interrupt Handling on x86

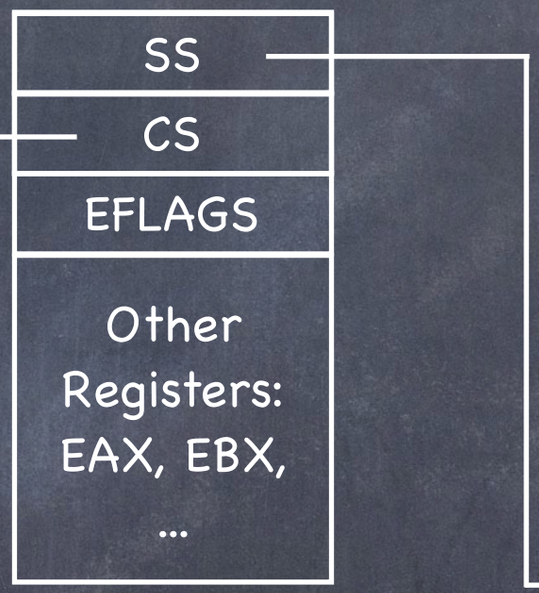
User-level Process

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

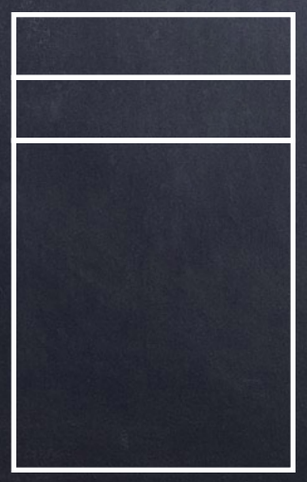
Interrupt Handling on x86

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

Interrupt Handling on x86

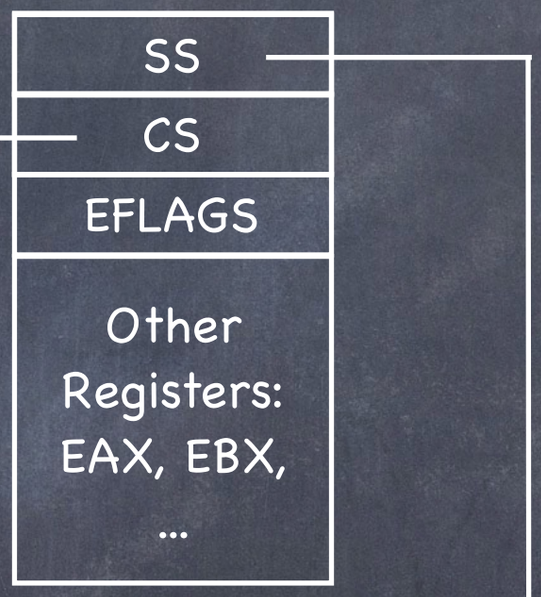
User-level Process

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)

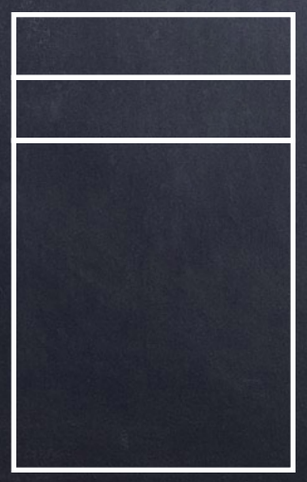
Interrupt Handling on x86

User-level Process

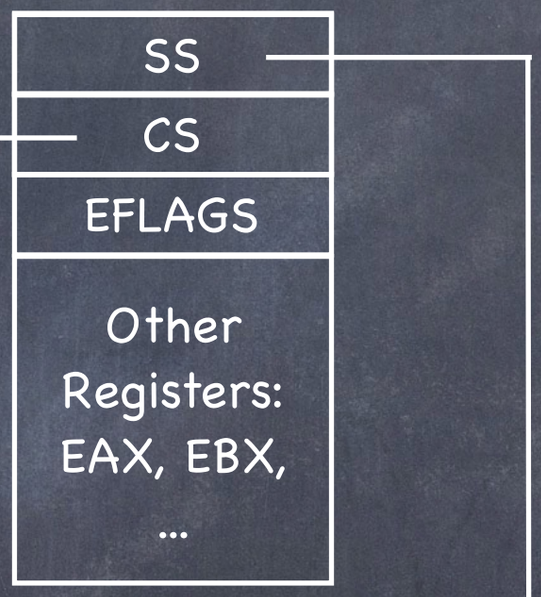
Code

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

Stack



Registers



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)

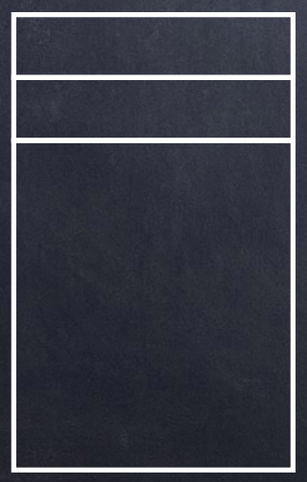
Interrupt Handling on x86

User-level Process

Code

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

Stack



Registers

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

Software (handler) performs this step

6. Save error code (optional)
7. Transfer control to interrupt handler
8. Handler pushes select registers on stack

Interrupt Handling on x86

User-level Process

Code

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

Stack

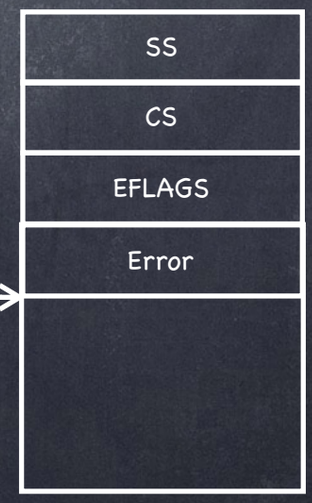
Registers

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

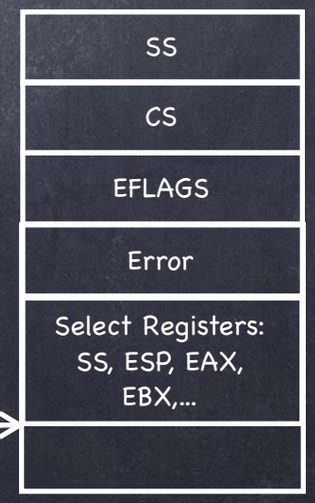
Registers

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 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 interrupts for I/O handling

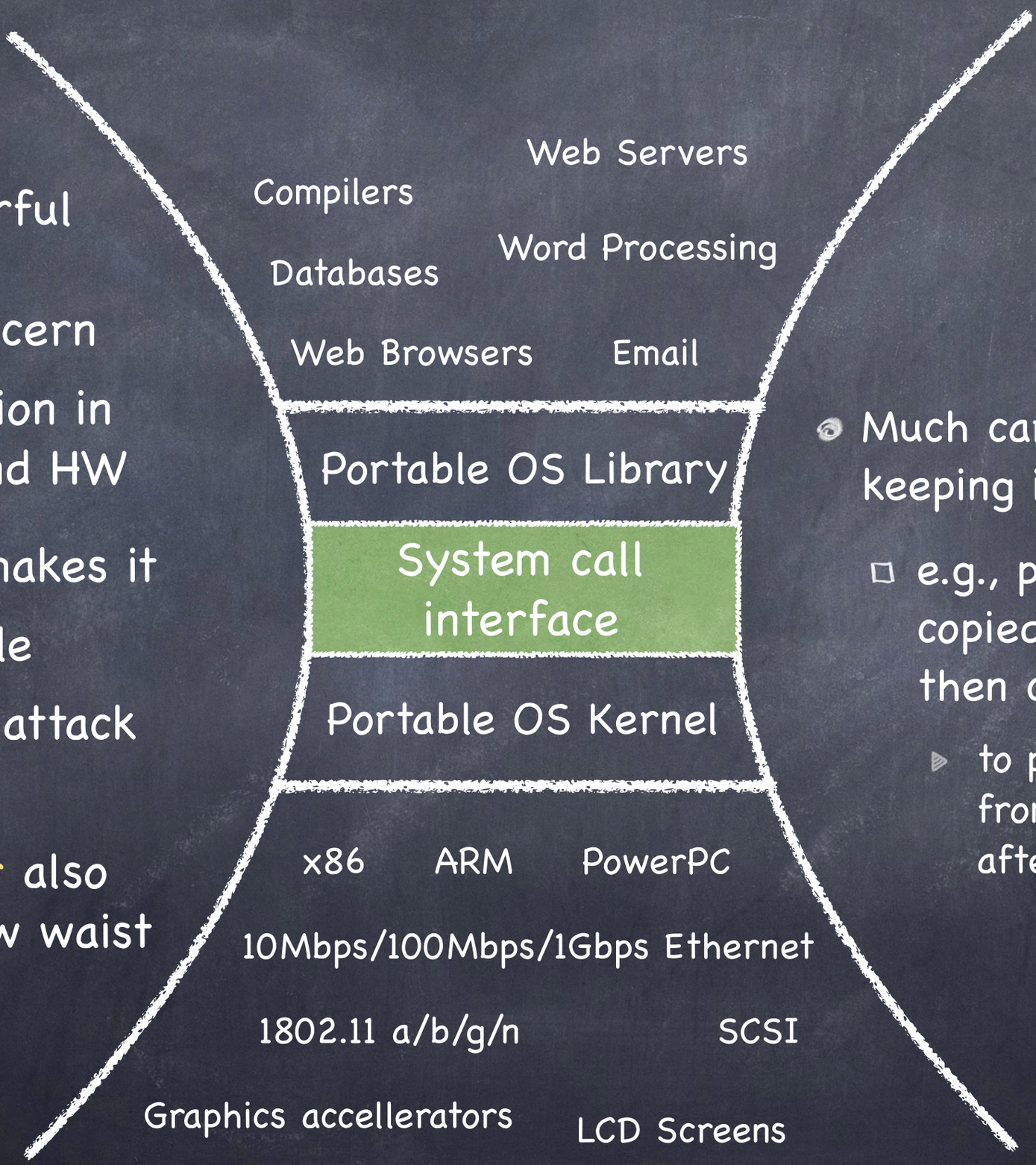
- This leads to potential race conditions
    - System's behavior depends on timing of uncontrollable events

System calls

- Programming interface to the services that the OS provides
  - Create new processes
  - Create/read/write/delete files
  - Send/receive data over sockets
  - …

The Narrow Waist

- 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 the narrow waist



- 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!

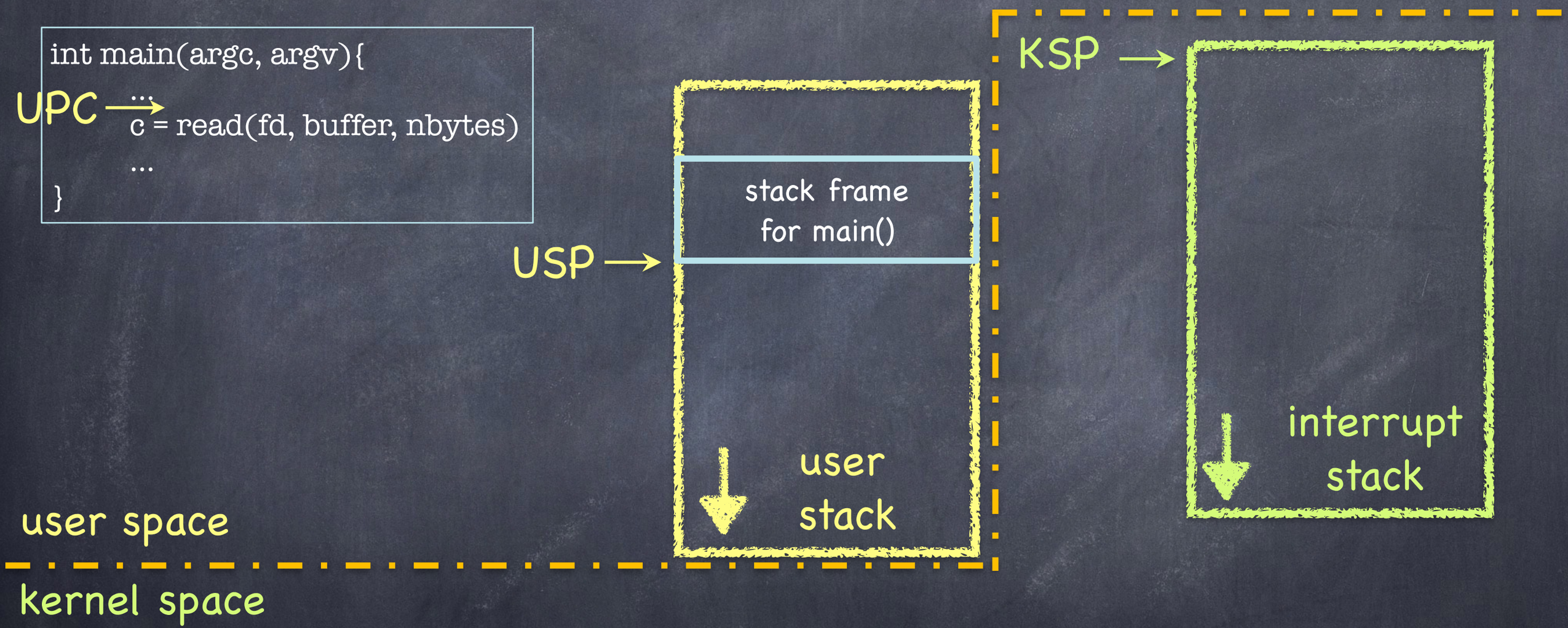
Executing a system call

- 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_handler`

- Process
    - Executes `return_from_function`

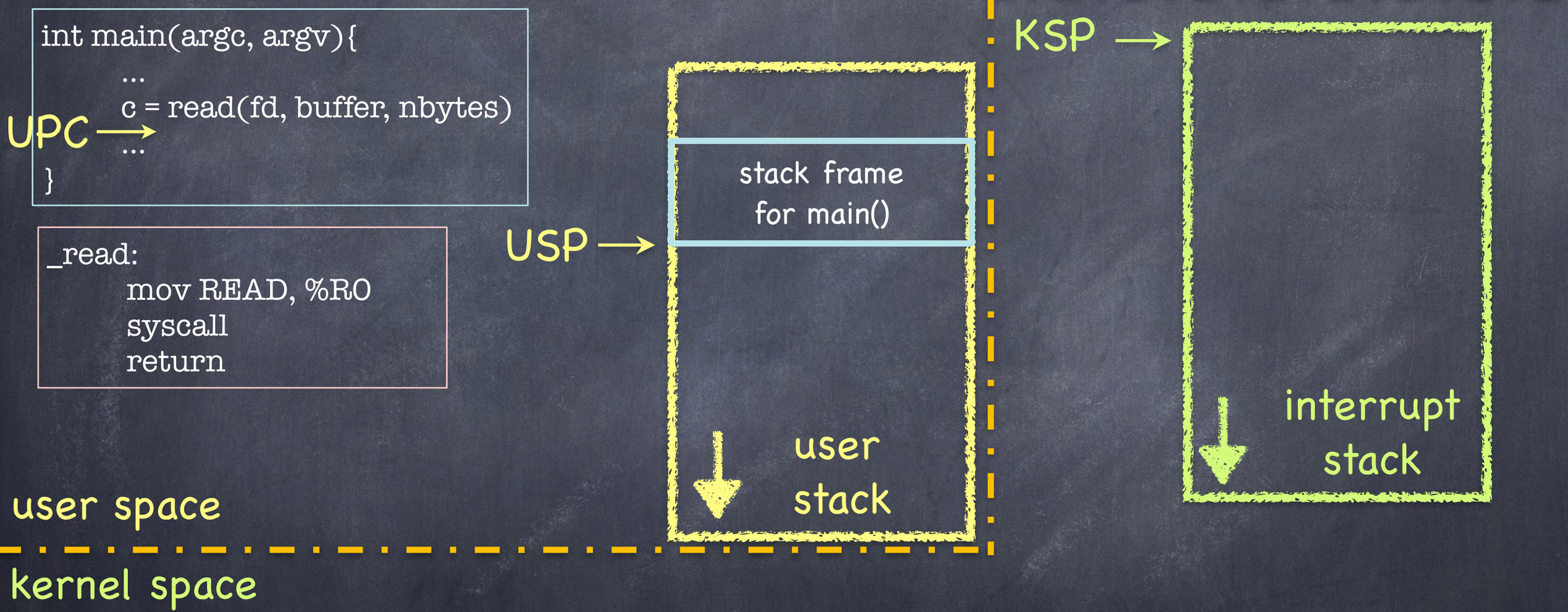
Executing read System Call



UPC: user program counter
USP: user stack pointer
KPC: kernel program counter
KSP: kernel stack pointer

note: interrupt stack is empty while process running

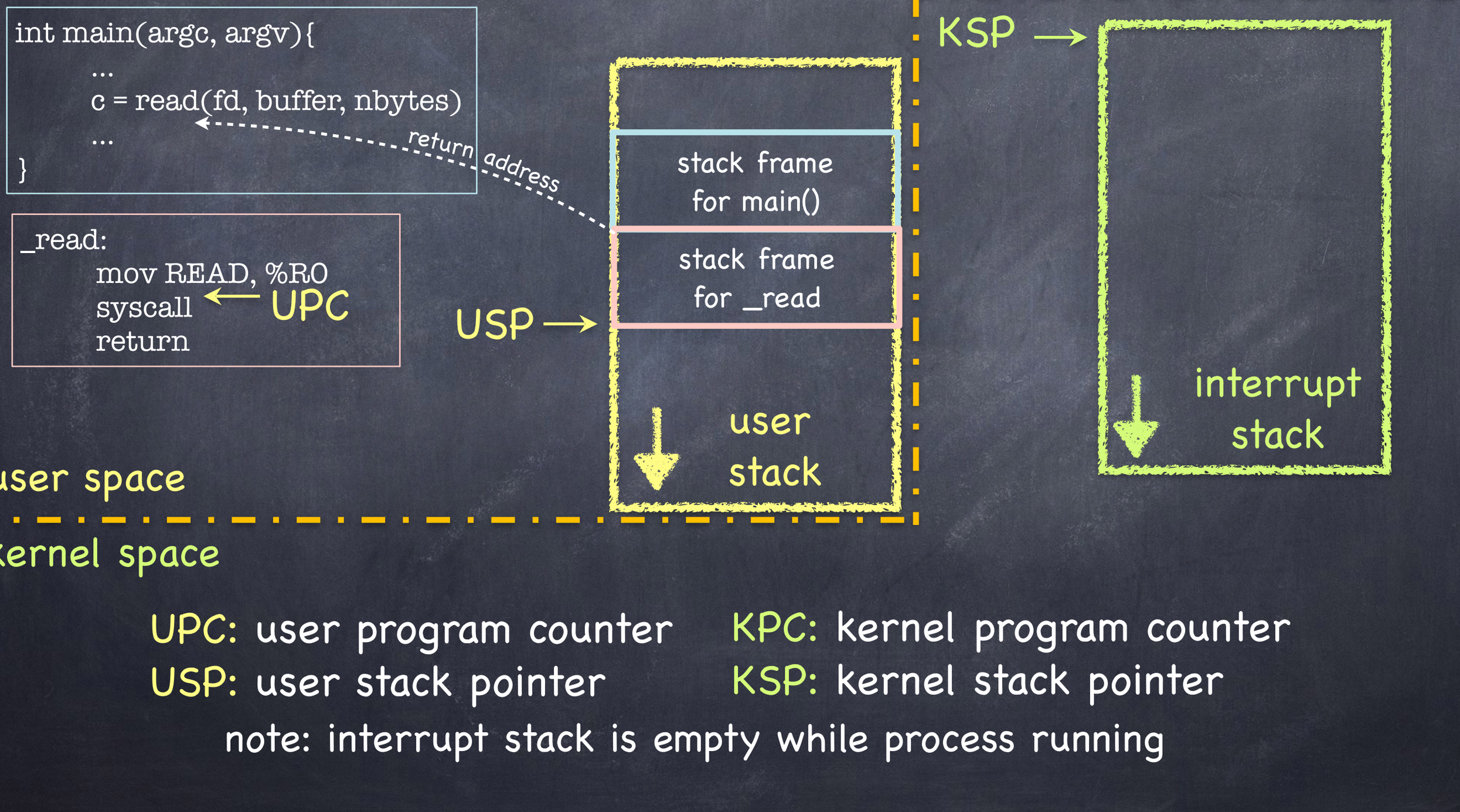
Executing read System Call



UPC: user program counter KPC: kernel program counter
USP: user stack pointer KSP: kernel stack pointer

note: interrupt stack is empty while process running

Executing read System Call



UPC: user program counter KPC: kernel program counter

USP: user stack pointer KSP: kernel stack pointer

note: interrupt stack is empty while process running

Executing read System Call

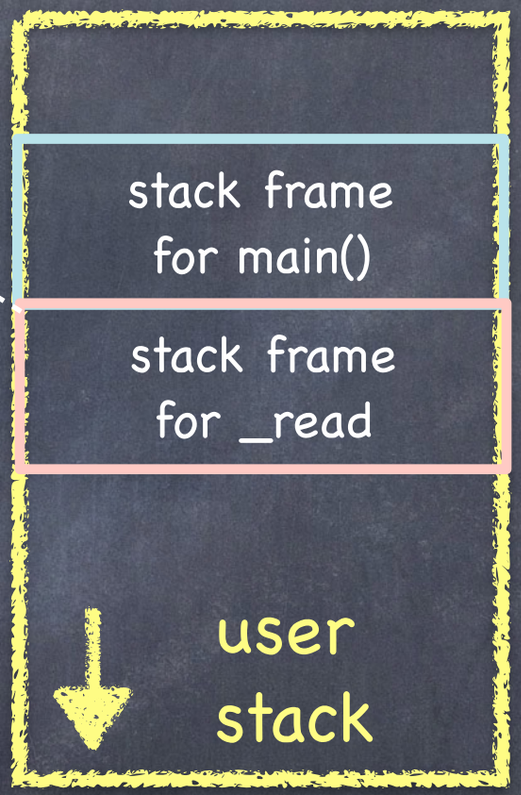
```
int main(argc, argv){
        ...
        c = read(fd, buffer, nbytes)
        ...
}
```

return address

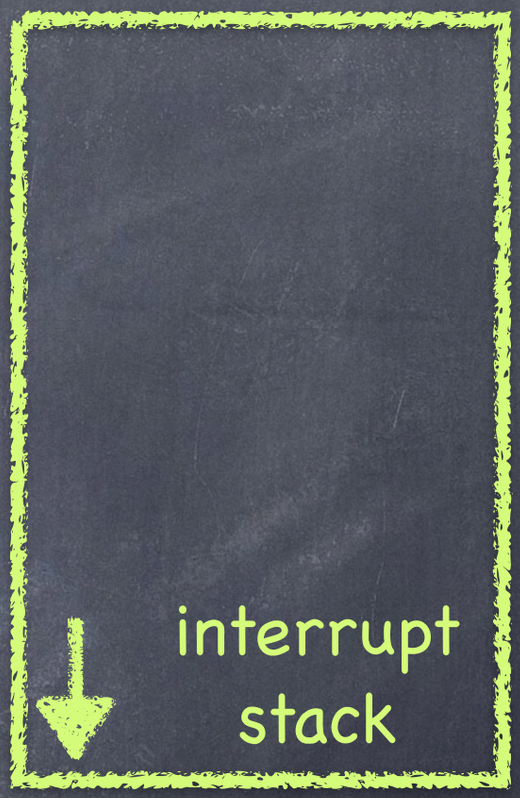
```
_read:
        mov READ, %R0
        syscall
        return
```

← UPC

USP →



KSP →



user space

kernel space

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

← KPC

Executing read System Call

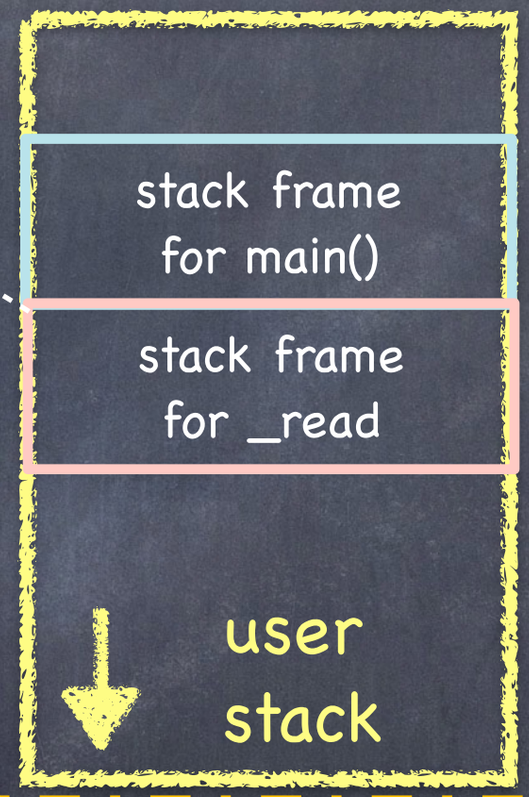
```
int main(argc, argv){
        ...
        c = read(fd, buffer, nbytes)
        ...
}
```

return address

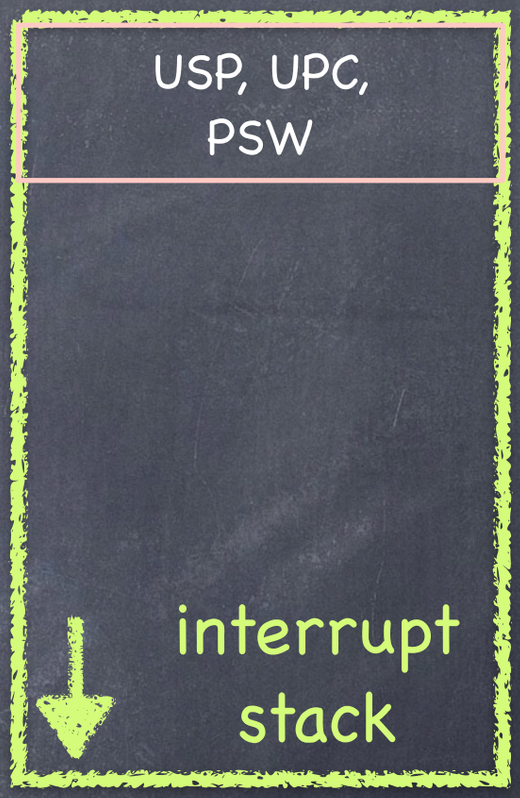
```
_read:
        mov READ, %R0
        syscall
        return
```

← UPC

USP →



KSP →



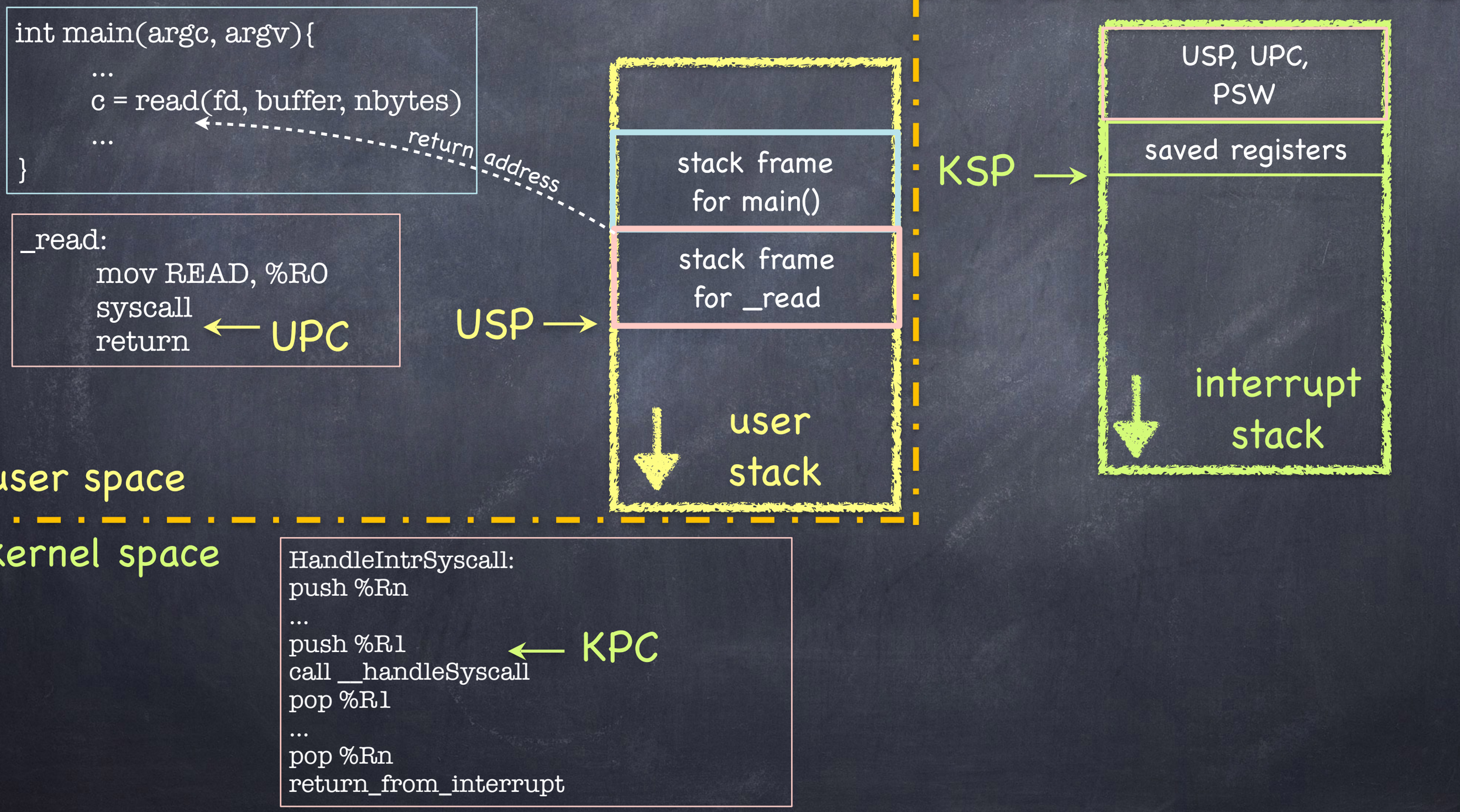
user space

kernel space

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

← KPC

Executing read System Call



Executing read System Call

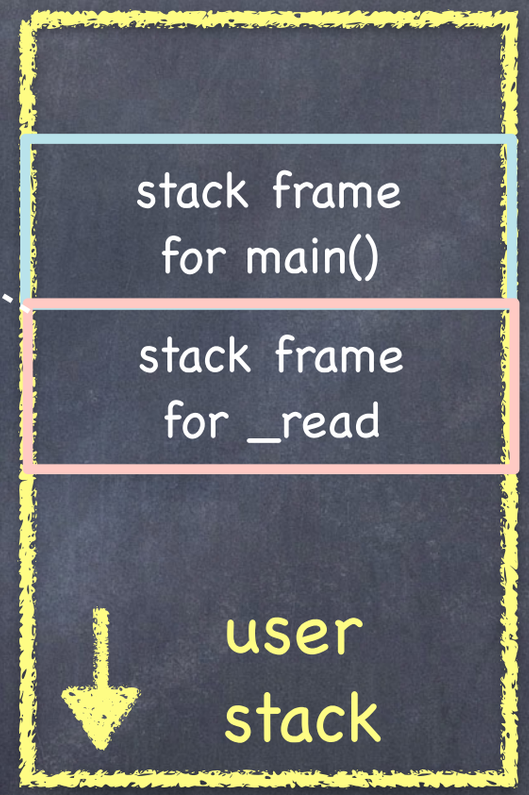
```
int main(argc, argv){
        ...
        c = read(fd, buffer, nbytes)
        ...
}
```

return address

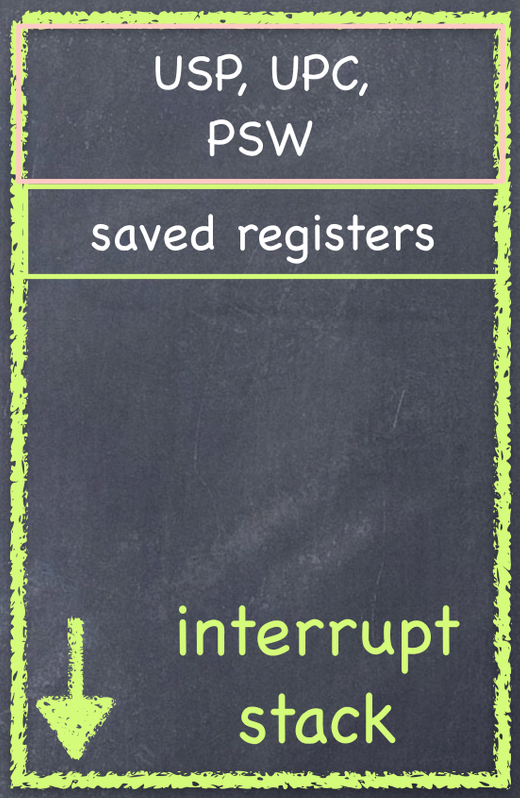
```
_read:
        mov READ, %R0
        syscall
        return
```

← UPC

USP →



KSP →



user space

kernel space

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

← KPC

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

Executing read System Call

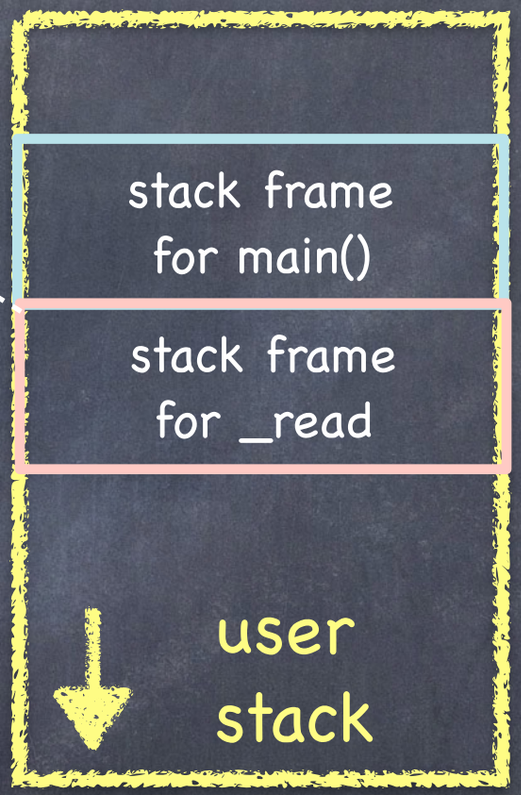
```
int main(argc, argv){
        ...
        c = read(fd, buffer, nbytes)
        ...
}
```

return address

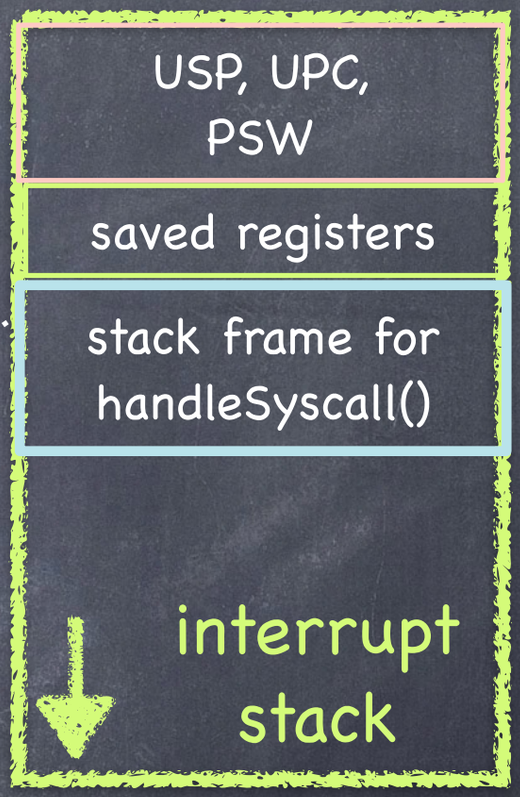
```
_read:
        mov READ, %R0
        syscall
        return
```

← UPC

USP →



KSP →



user space

kernel space

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

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

← KPC

Signals

- **Signals are *virtualized* Interrupts**
  - Asynchronous notifications in user space
  - Some examples:

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., CTRL-C from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated
20	SIGSTP	Stop until SIGCONT	Stop signal from terminal (e.g., CTRL-Z from keyboard)

Receiving a signal

- **Each signal prompts one of these default actions**
  - Terminate the process
  - Ignore the signal
  - Terminate the process and dump core
  - Stop the process
  - Continue the process, if stopped

- **Signal can be “caught” by executing a user-level function called signal handler**
  - Similar to exception handler invoked in response to an asynchronous interrupt

- **Process could also be suspended waiting for a signal to be caught**
  - (Synchronously)

Context switch

Multiple concurrent processes

- **How to yield from one process to another?**

- "switch" from executing the Current process to some other READY process
    - Current process: RUNNING—>READY
    - Next process: READY—>RUNNING

- Steps involved:
    - Save kernel registers of Current on its interrupt stack
    - Save kernel stack pointer of Current in it PCB
    - Restore kernel stack pointer of Next from its PCB
    - Restore kernel registers of Next from its interrupt stack

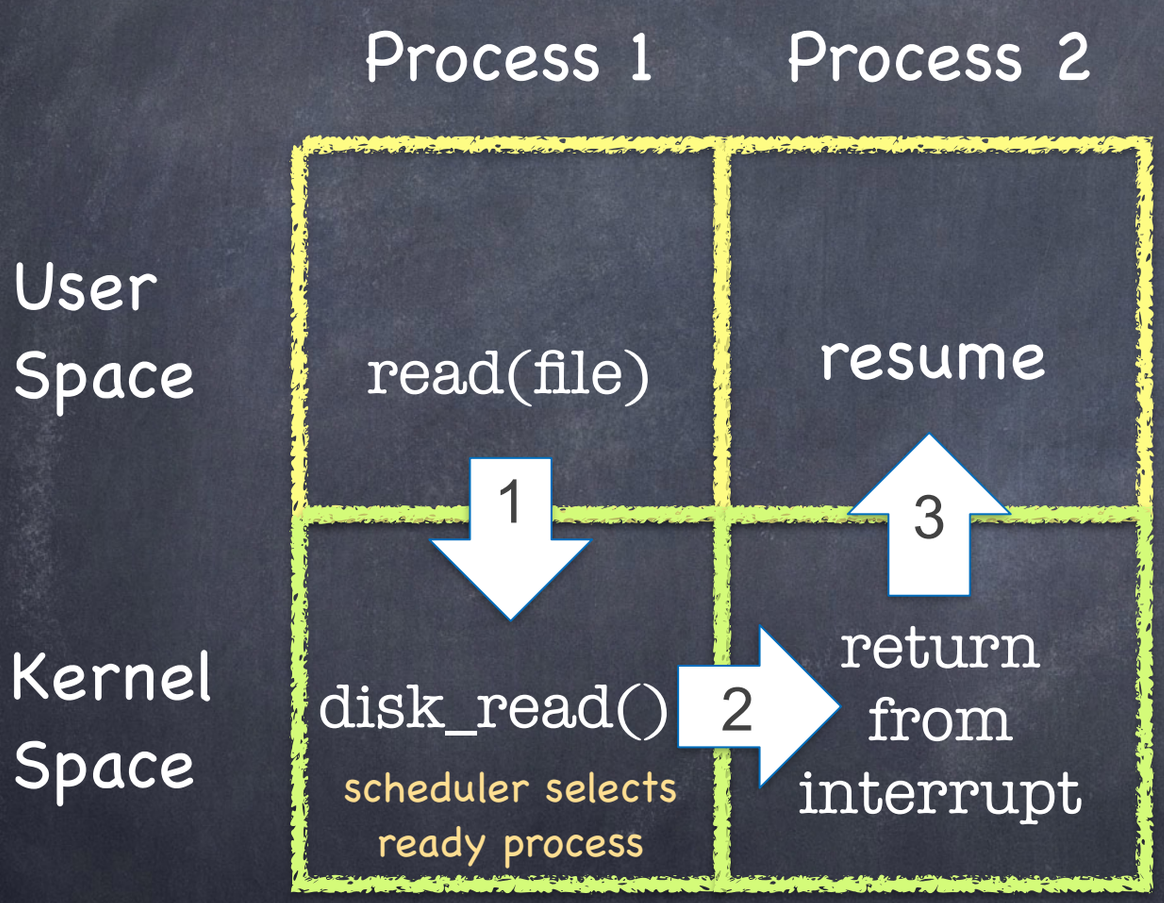
Three flavors of context switch

- **Interrupt:** from user to kernel space
  - On system call, exception, or interrupt
  - Stack switch: process X user stack —> process X interrupt stack

- **Yield:** between two processes, inside kernel
  - From one PCB/interrupt stack to another
  - Stack switch: process X interrupt stack —> process Y interrupt stack

- **Return from interrupt:** from kernel to user space
  - Stack switch: process X interrupt stack —> process X user stack

Switching between Processes



1. Save Process 1 user registers
2. Save Process 1 kernel registers and restore Process 2 kernel registers
3. Restore Process 2 user registers

We are ready to understand the life cycle of a process!

Process Life Cycle

Init

Ready

Running

Zombie

Waiting

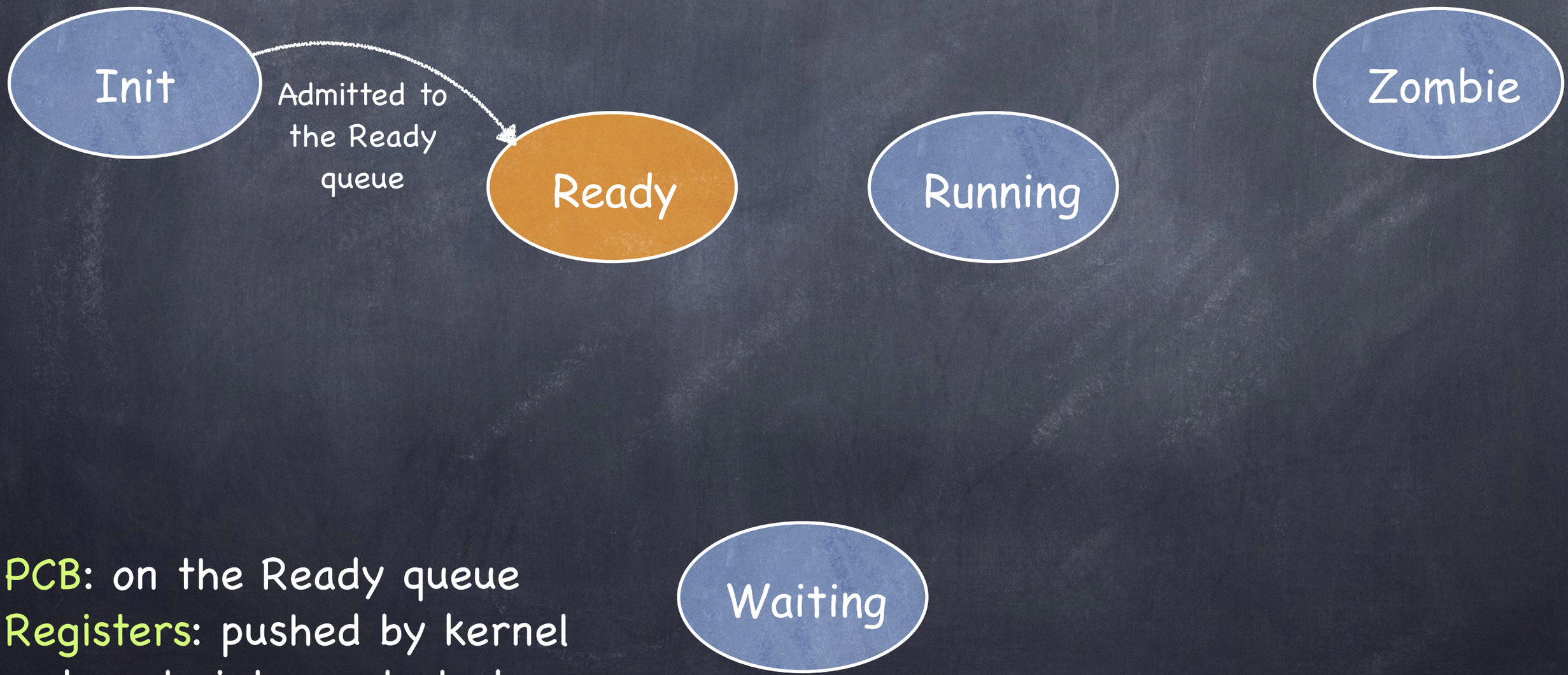
Process Life Cycle

PCB: being created
Registers: uninitialized

Process Life Cycle

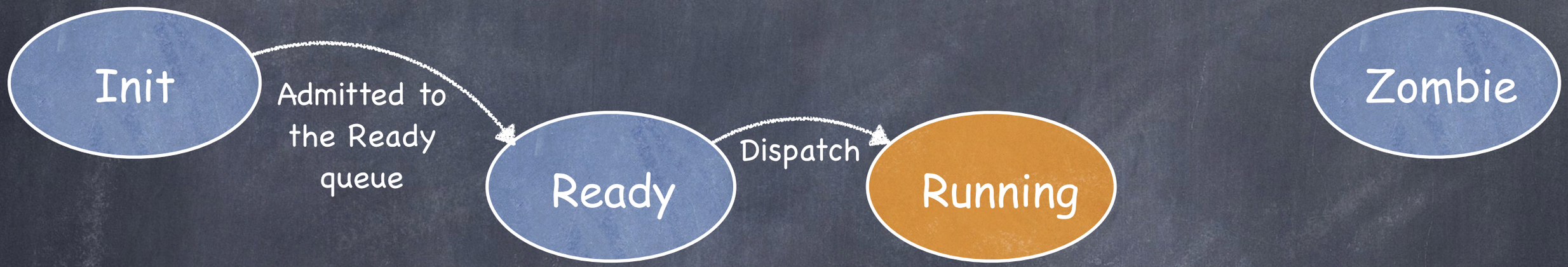
PCB: being created
Registers: uninitialized

Process Life Cycle



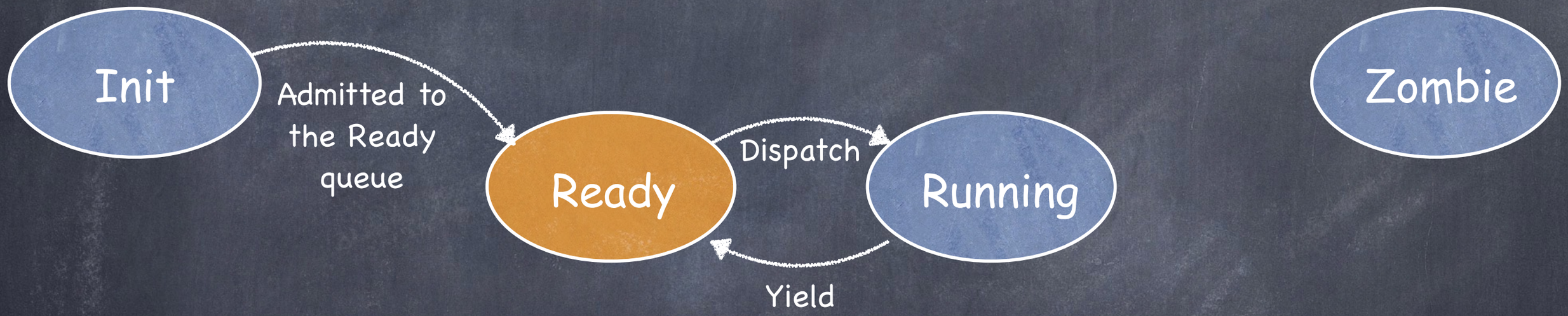
PCB: on the Ready queue
Registers: pushed by kernel code onto interrupt stack

Process Life Cycle



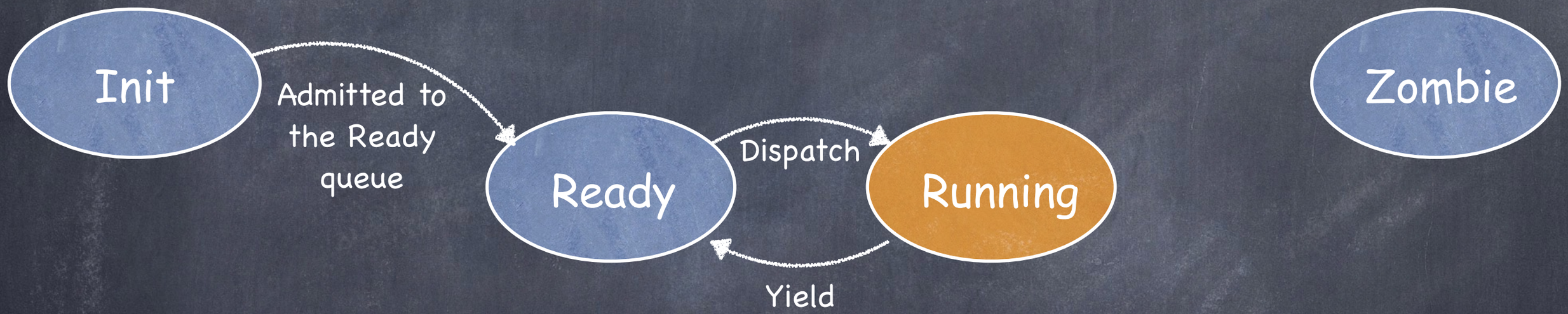
PCB: currently executing
Registers: popped from interrupt stack into CPU

Process Life Cycle



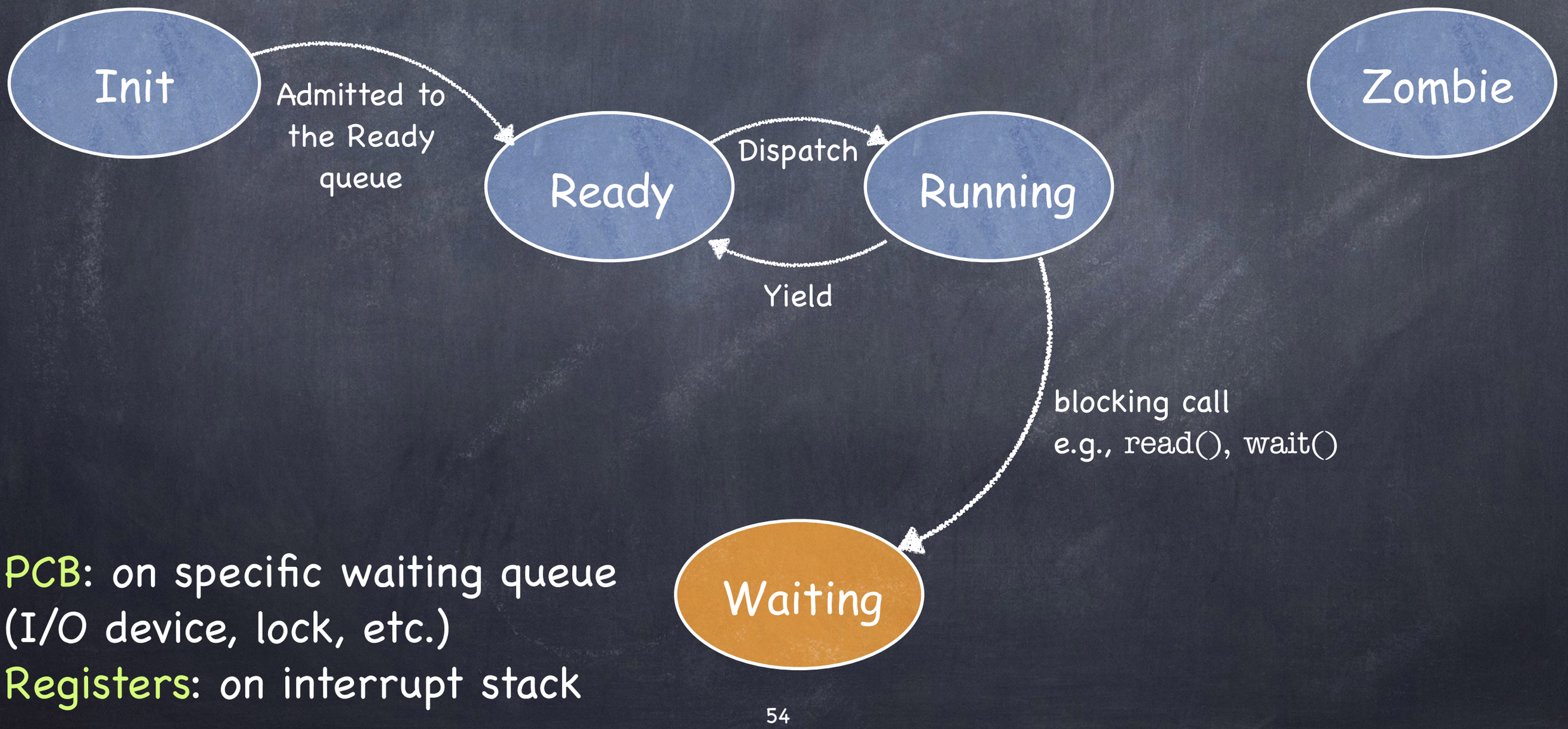
PCB: on Ready queue
Registers: pushed onto interrupt stack (SP saved in PCB)

Process Life Cycle



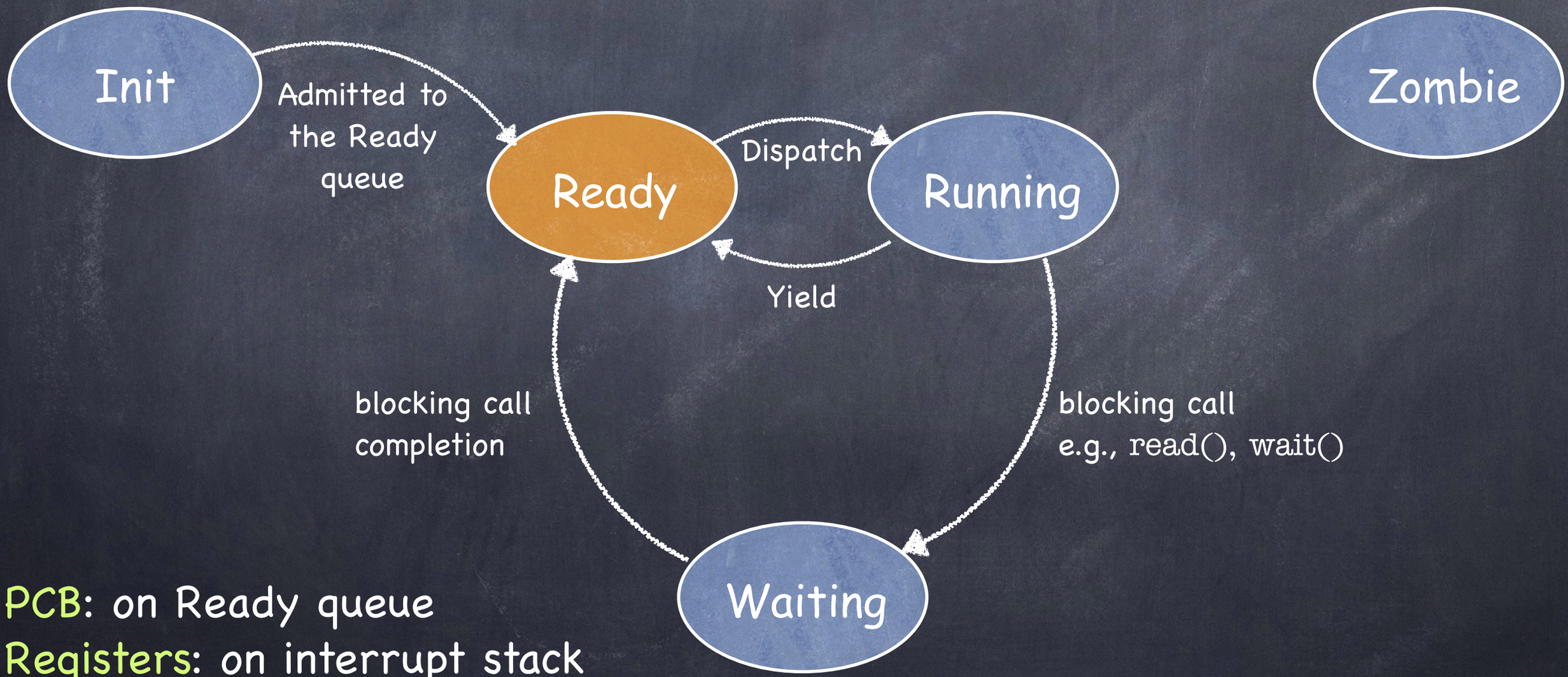
PCB: currently executing
Registers: SP restored from PCB; others restored from stack

Process Life Cycle



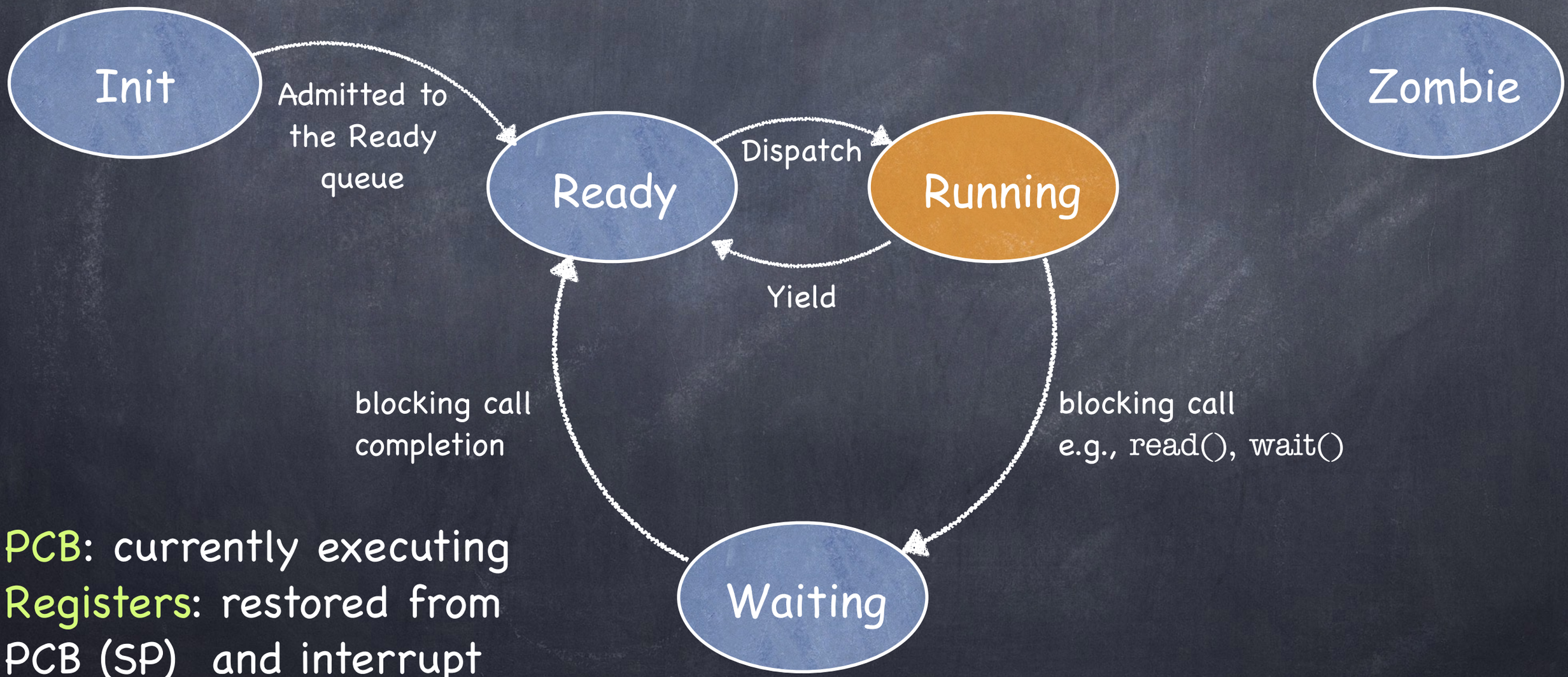
PCB: on specific waiting queue (I/O device, lock, etc.)
Registers: on interrupt stack

Process Life Cycle



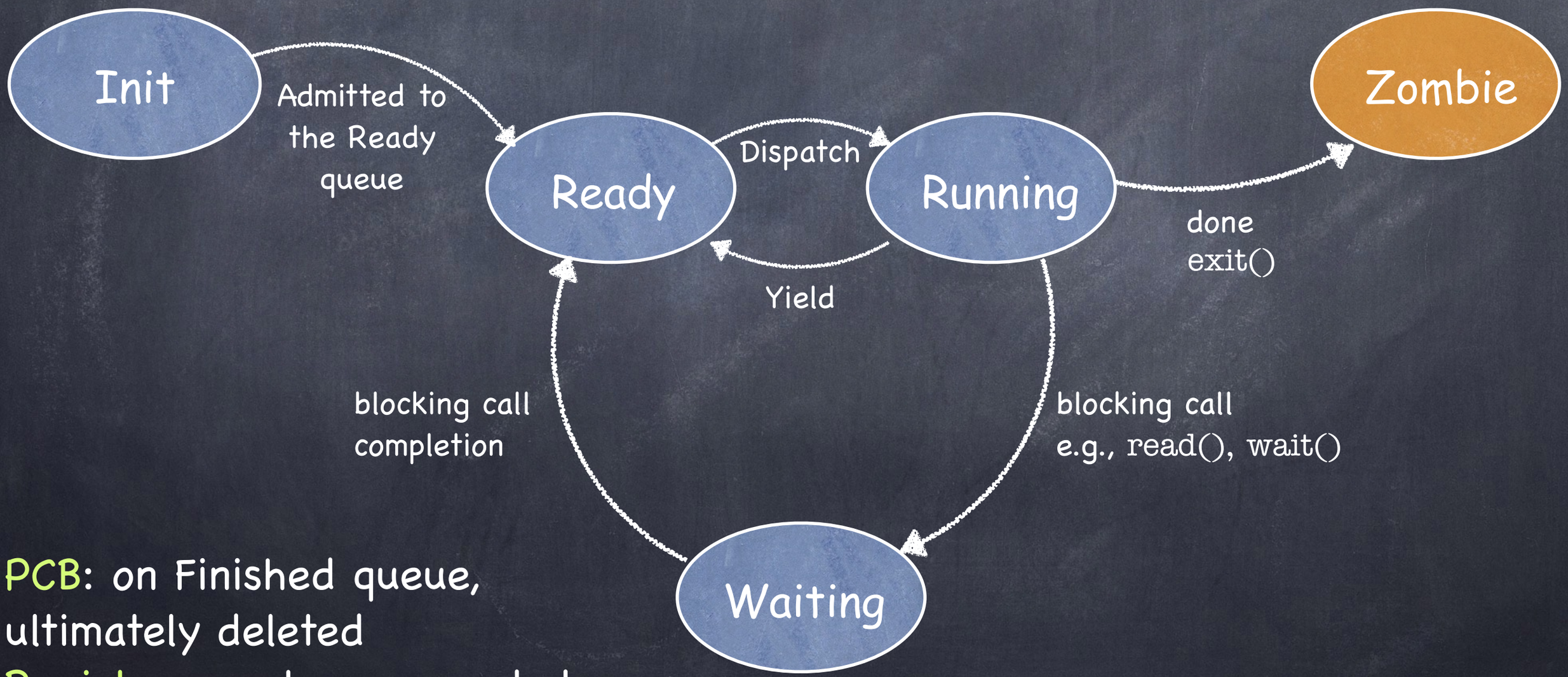
PCB: on Ready queue
Registers: on interrupt stack

Process Life Cycle



PCB: currently executing
Registers: restored from PCB (SP) and interrupt stack into CPU

Process Life Cycle



PCB: on Finished queue, ultimately deleted

Registers: no longer needed

Booting an OS

- **Steps in booting an OS**
  - CPU starts at fixed address
    - In supervisor mode, with interrupts disabled
  - BIOS (in ROM) loads “boot loader” code into memory and runs it
    - From specified storage or network device
  - Boot loader loads OS kernel code into memory and runs it

- **Initialization**
  - Determine location/size of physical memory
  - Set up initial page tables
  - Initialize the interrupt vector
  - Determine which devices the computer has
  - Initialize “file system” code
  - Load first process from file system
  - Start first process