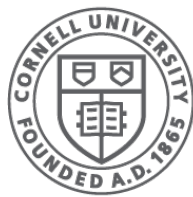


# Processes

(Chapters 3-6)

CS 4410  
Operating Systems



**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

[R. Agarwal, L. Alvisi, A. Bracy, M. George  
Fred B. Schneider, E. Sirer, R. Van Renesse]

# Before we start

- Registers
  - general-purpose vs device
- Interrupts in user space
  - what happens to the PC?
- kernel stack == interrupt stack
- zoom recordings
- tooth
- Harmony installation

# Process vs Program

- A program consists of code and data
  - specified in some programming language
- Typically stored in a file on disk
- “*Running a program*” = creating a process
  - you can run a program multiple times!
    - one after another or even concurrently

# What is an “*Executable*”?

An executable is a **file** containing:

- executable code
  - CPU instructions
- data
  - information manipulated by these instructions
- Obtained by *compiling* a program
  - and linking with libraries

# What is a “*Process*”?

- An executable running on an **abstraction** of a computer:
  - Address Space (memory) +  
Execution Context (registers incl. PC and SP)
    - manipulated through machine instructions
  - Environment (clock, files, network, ...)
    - manipulated through system calls

## *A good abstraction:*

- is portable and hides implementation details
- has an intuitive and easy-to-use interface
- can be instantiated many times
- is efficient to implement

# Process $\neq$ Program

A program is passive:  
code + data

A process is *alive*:  
mutable data + registers + files + ...

Same program can be run multiple time  
simultaneously (1 program, 2 processes)

```
> ./program &  
> ./program &
```

# A Day in the Life of a Program

Compiler

(+ Assembler + Linker)

Loader

*"It's alive!"*

sum.c

sum

pid xxx

source files

executable

process

```
#include <stdio.h>
```

```
int max = 10;
```

```
int main () {  
    int sum = 0;  
    add(max, &sum);  
    printf("%d", sum);  
    ...  
}
```

...	0C40023C
21035000	
1b80050c	
8C048004	
21047002	
0C400020	
...	
10201000	
21040330	
22500102	
...	

**.text** **main**

**.data** **max**



0xffffffff

stack

heap

max data

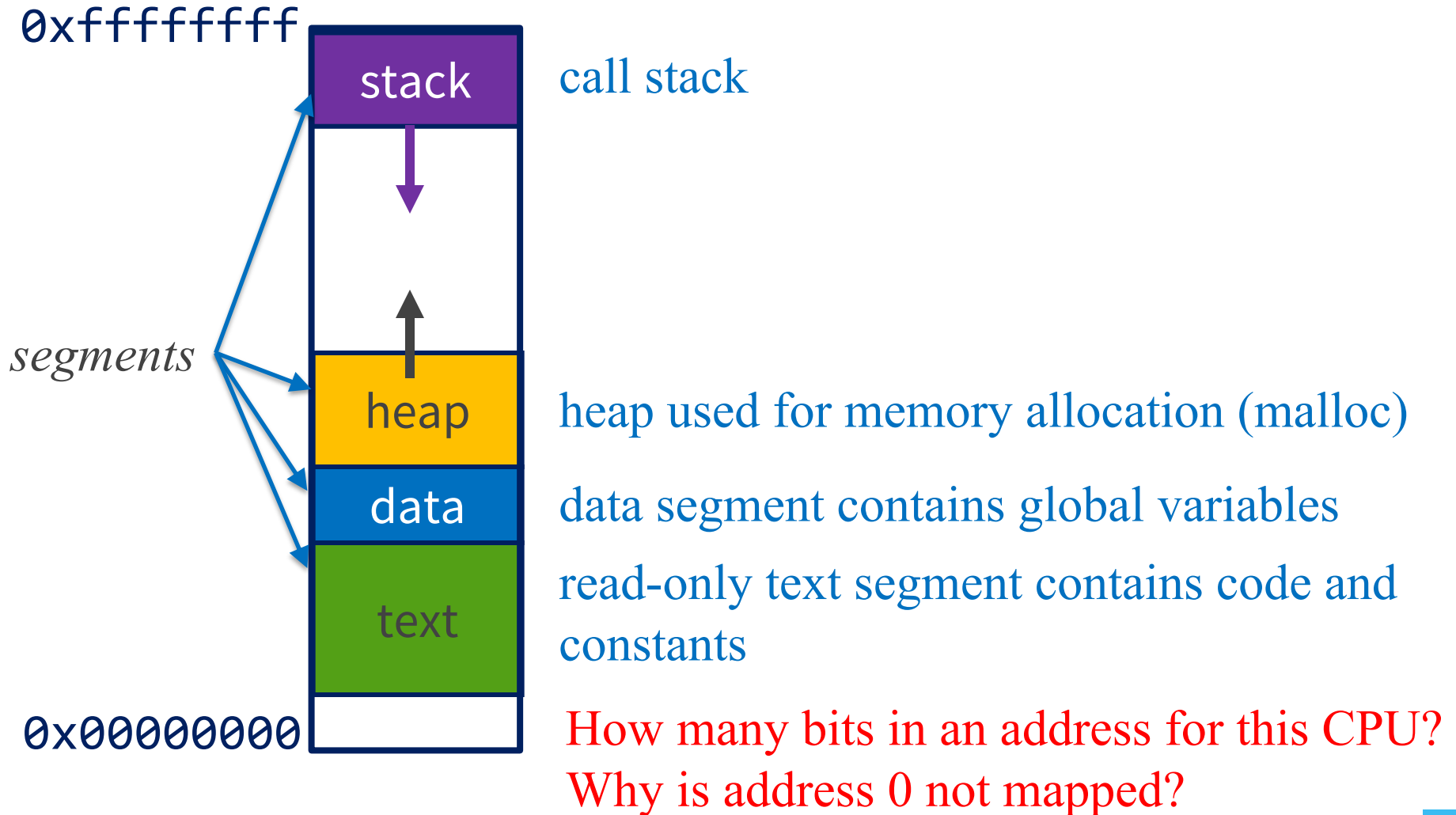
jal text  
addi

0x10000000

0x00400000

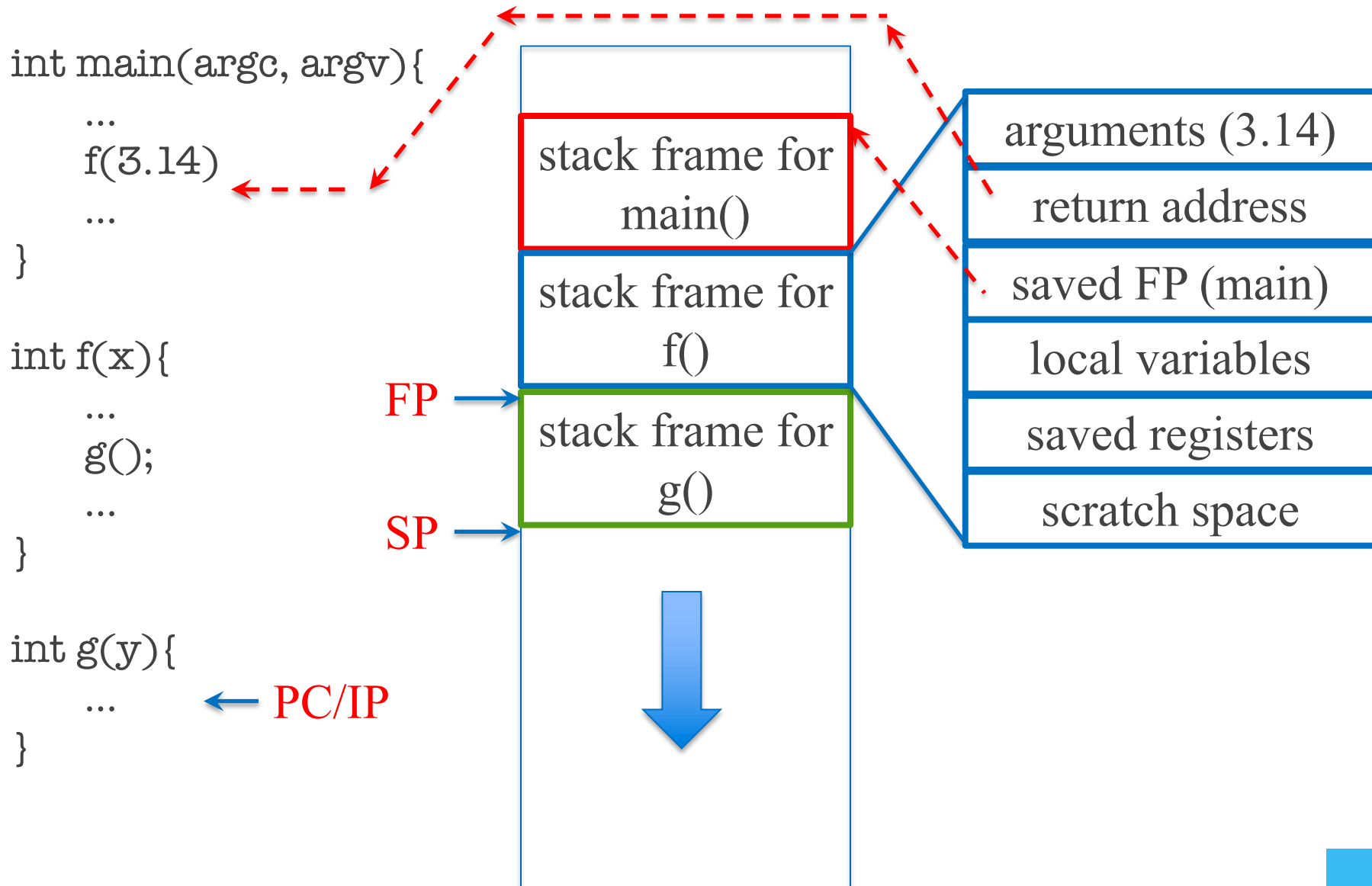
0x00000000

# Logical view of process memory

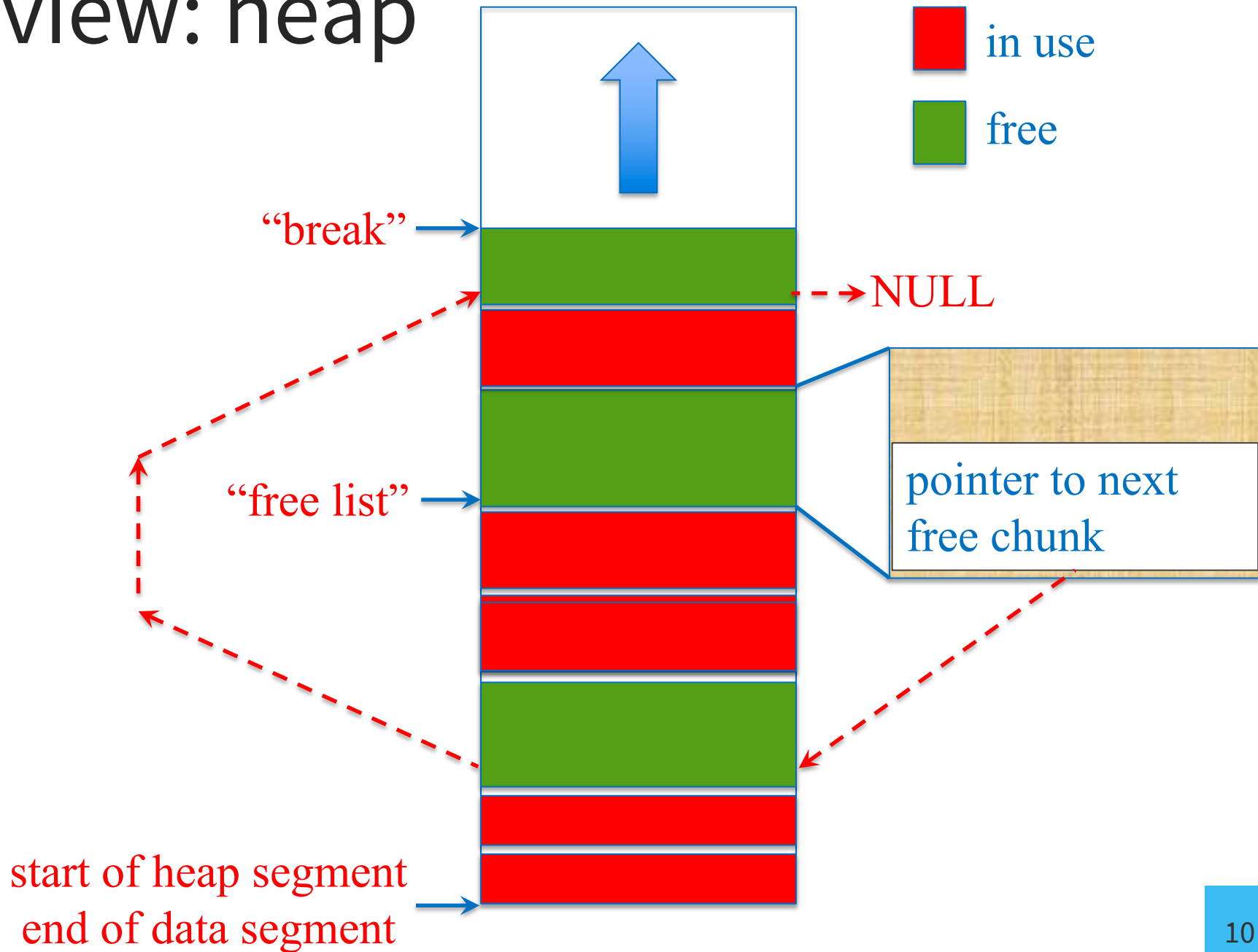




# Review: stack (aka call stack)



# Review: heap



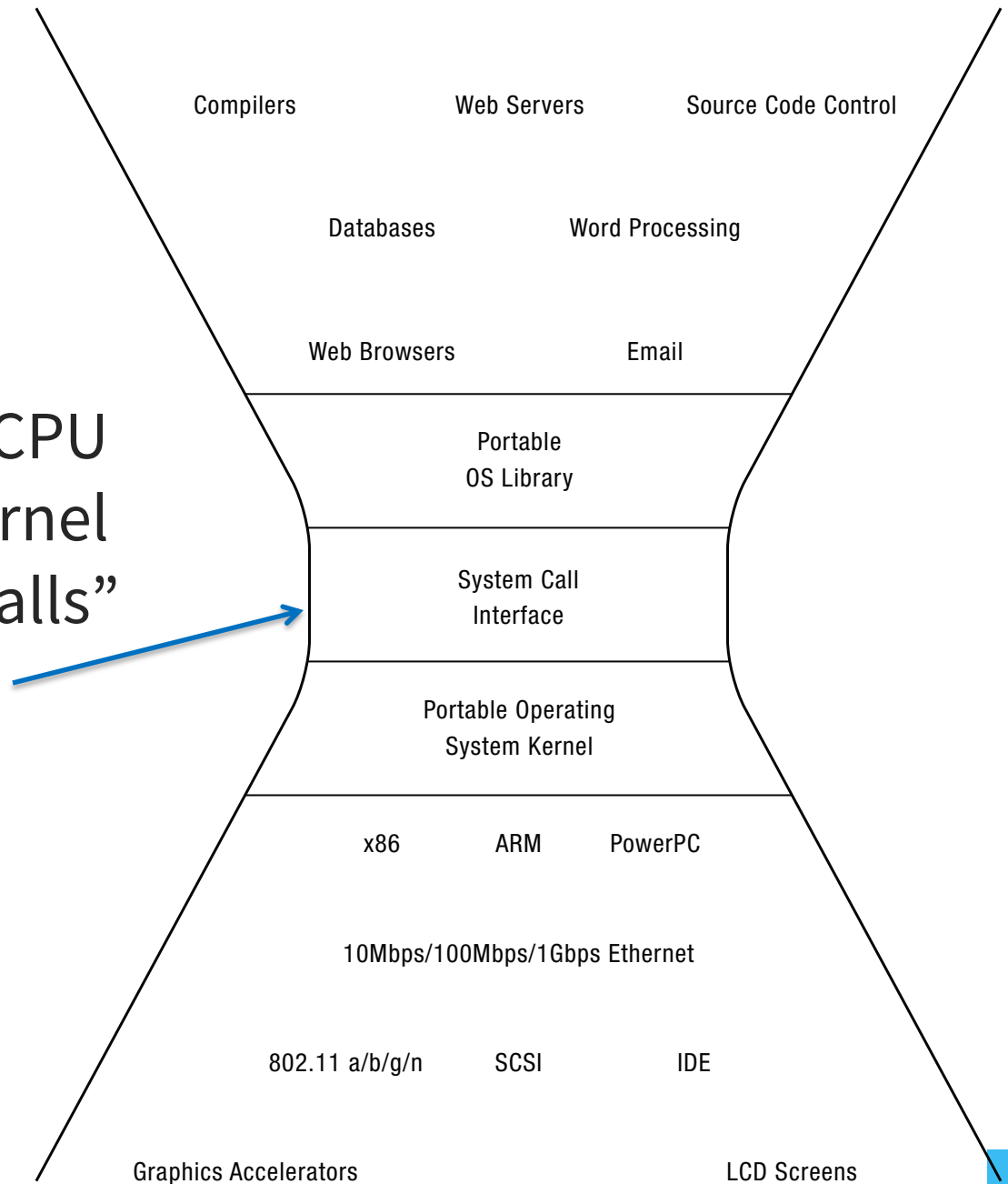
# Environment

- CPU, registers, memory allow you to implement algorithms
- But how do you
  - ☐ read input / write to screen
  - ☐ create/read/write/delete files
  - ☐ create new processes
  - ☐ send/receive network packets
  - ☐ get the time / set alarms
  - ☐ terminate the current process



# System Calls

- A process runs on CPU
- Can access O.S. kernel through “system calls”
- *Skinny interface*
  - Why?



# Why a “skinny” interface?

- Portability
  - easier to implement and maintain
  - e.g., many implementations of “Posix” interface
- Security
  - “small attack surface”: easier to protect against vulnerabilities

*not just the O.S. interface. Internet “IP” layer is another good example of a skinny interface*

# Executing a system call

## Process:

1. Calls system call function in library
2. Places arguments in registers and/or pushes them onto user stack
3. Places syscall type in a dedicated register
4. Executes **syscall** machine instruction

## Kernel:

5. Executes **syscall** interrupt handler
6. Places result in dedicated register
7. Executes **return\_from\_interrupt**

## Process:

8. Executes **return\_from\_function**

# Executing read System Call

```
int main(argc, argv){  
    ...  
    read(f) ← PC  
    ...  
}
```

SP →

stack frame for  
main()



user stack



interrupt  
stack

user space

kernel space

note interrupt stack empty while process running

# Executing read System Call

```
int main(argc, argv){  
    ...  
    read(f)  
    ...  
}
```

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

SP

stack frame for  
main()

stack frame for  
\_read

user stack

interrupt  
stack

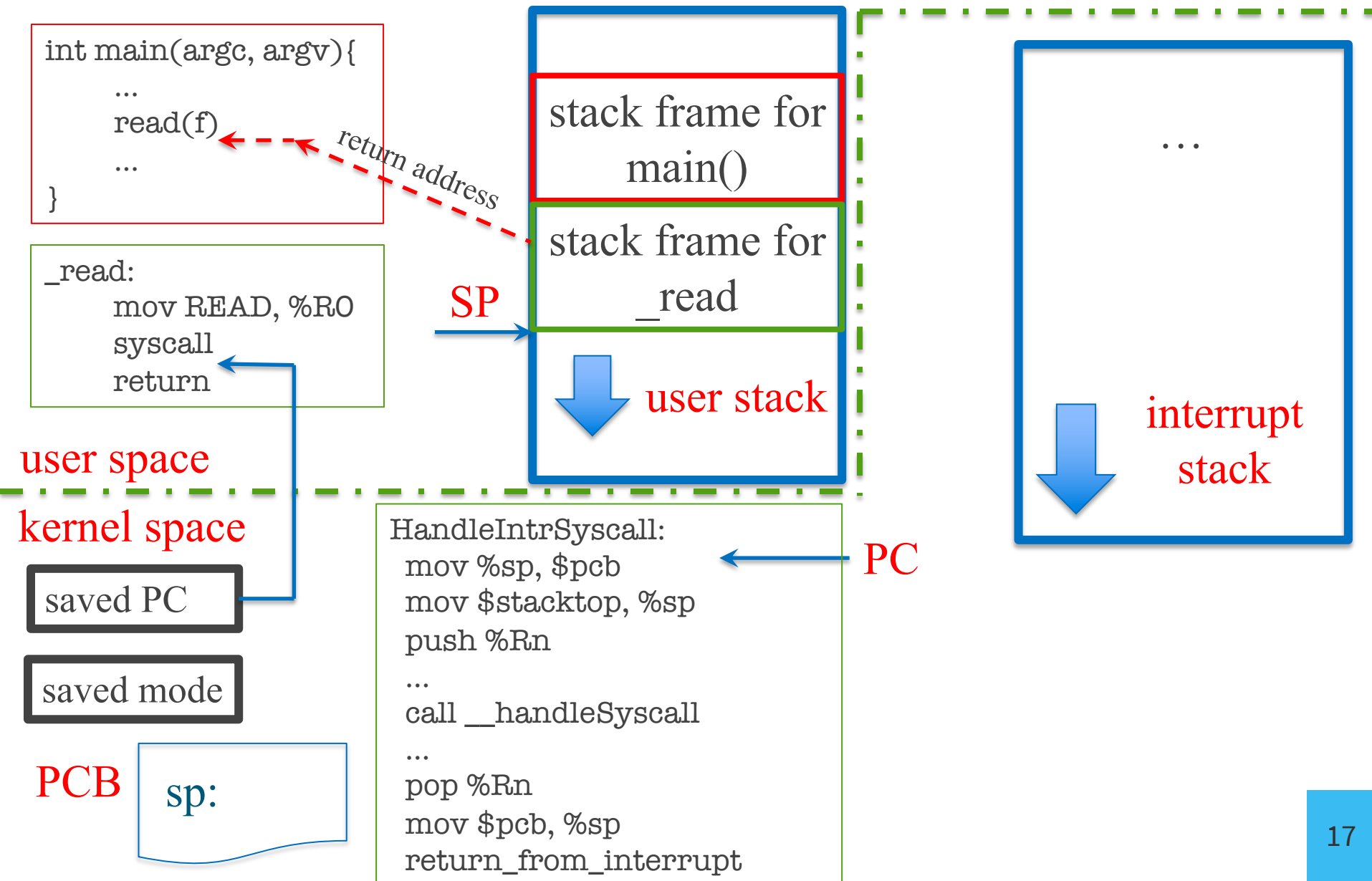
user space

kernel space

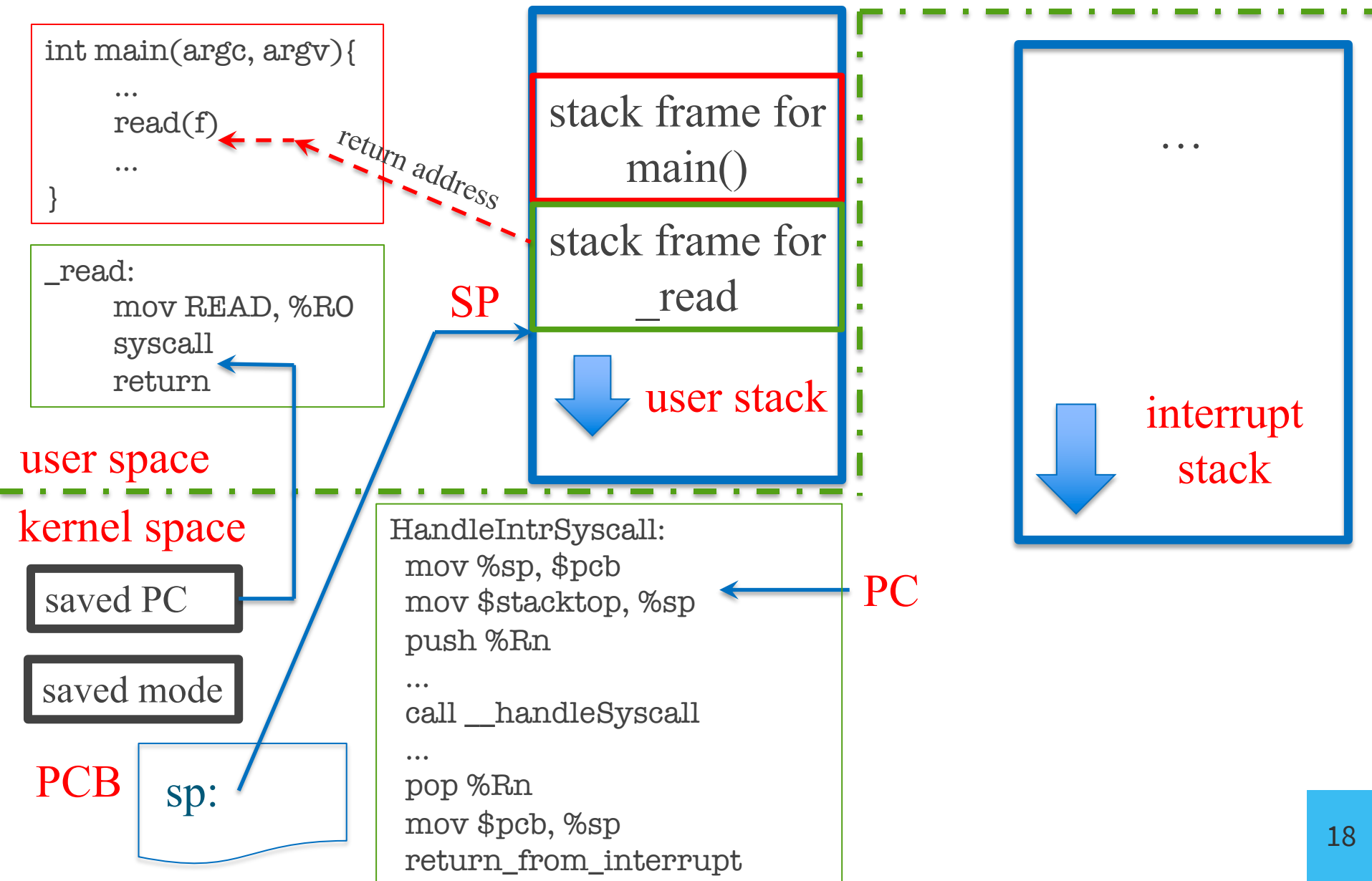
note interrupt stack empty while process running



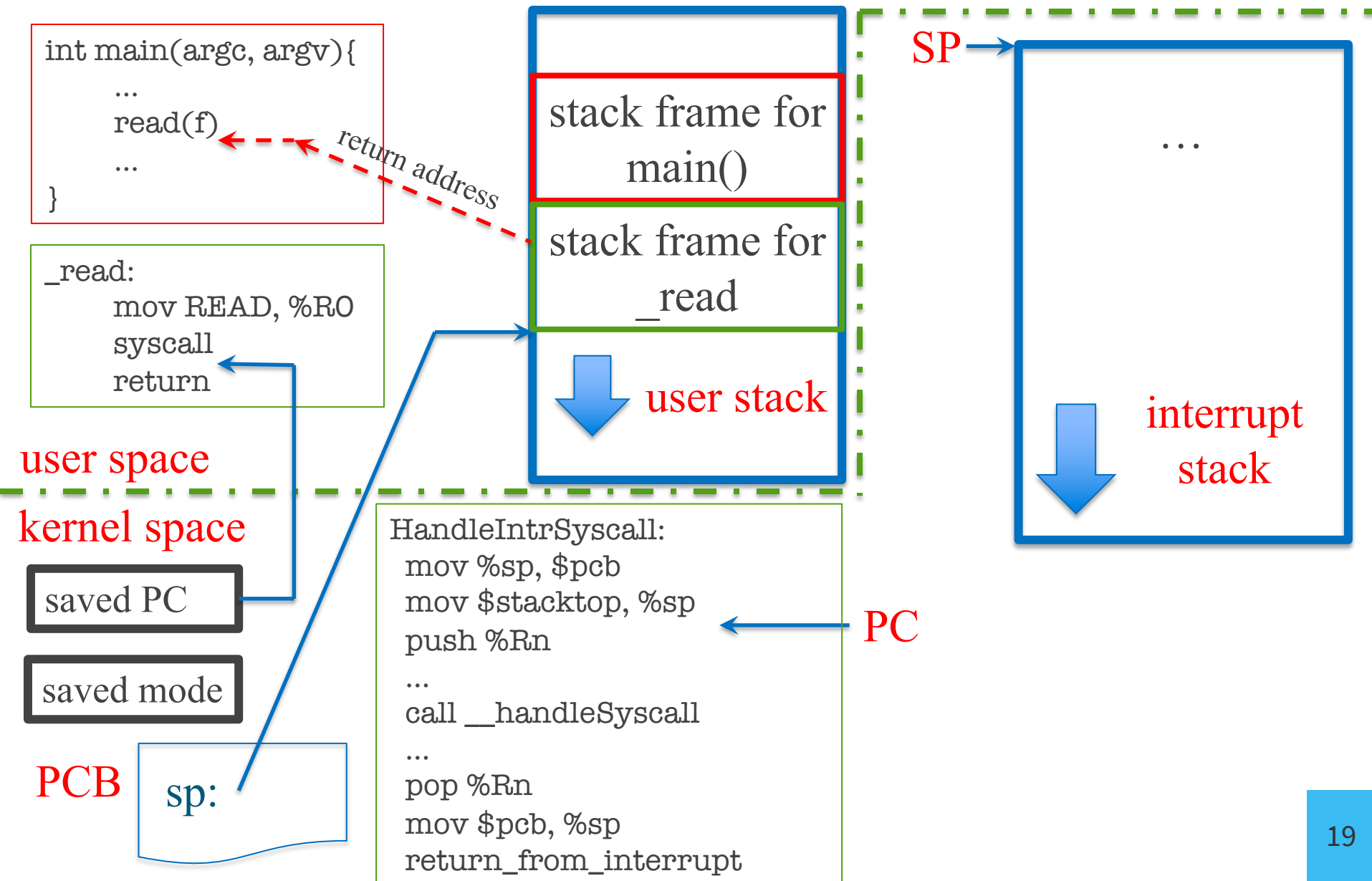
# Executing read System Call



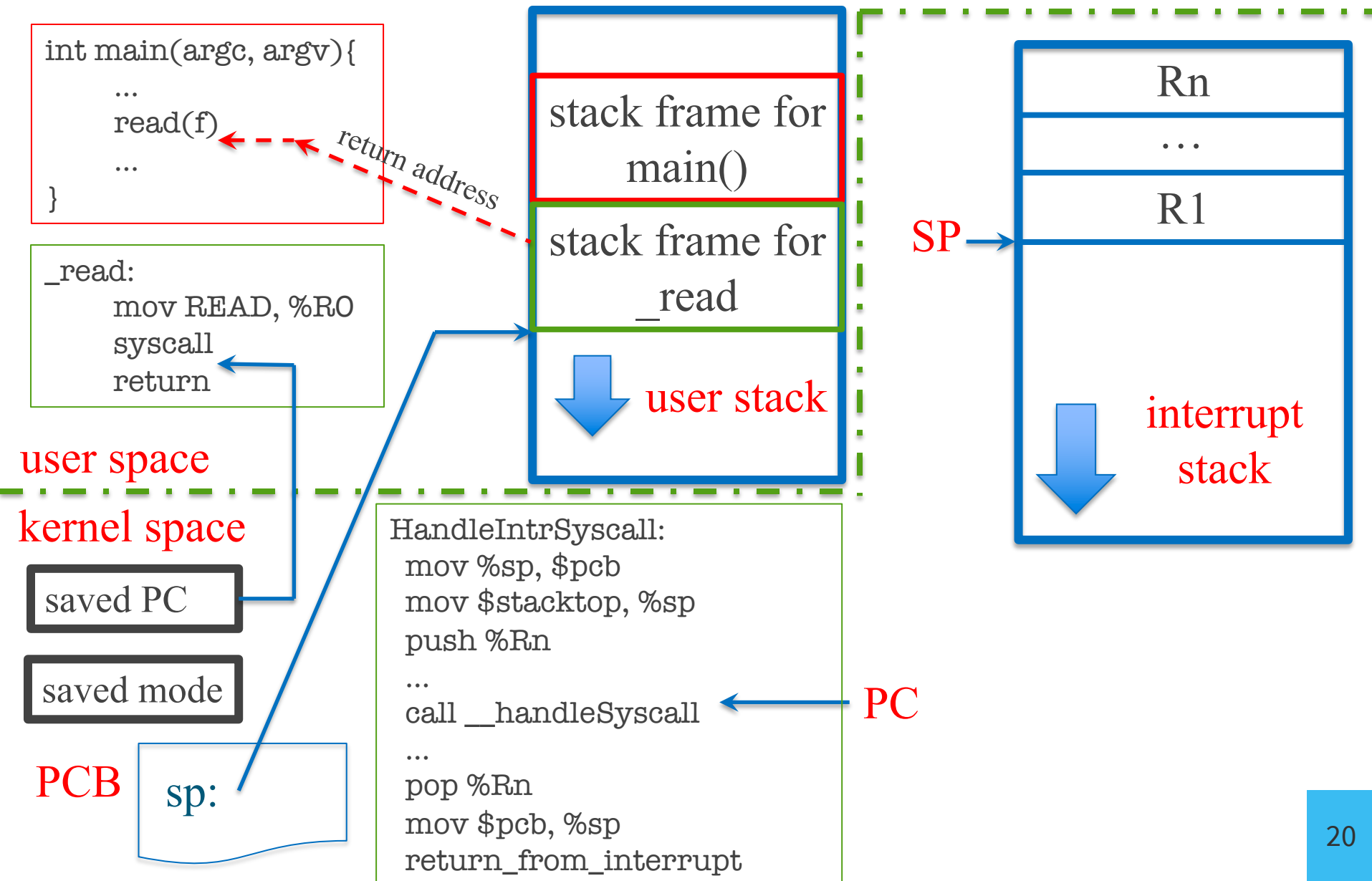
# Executing read System Call



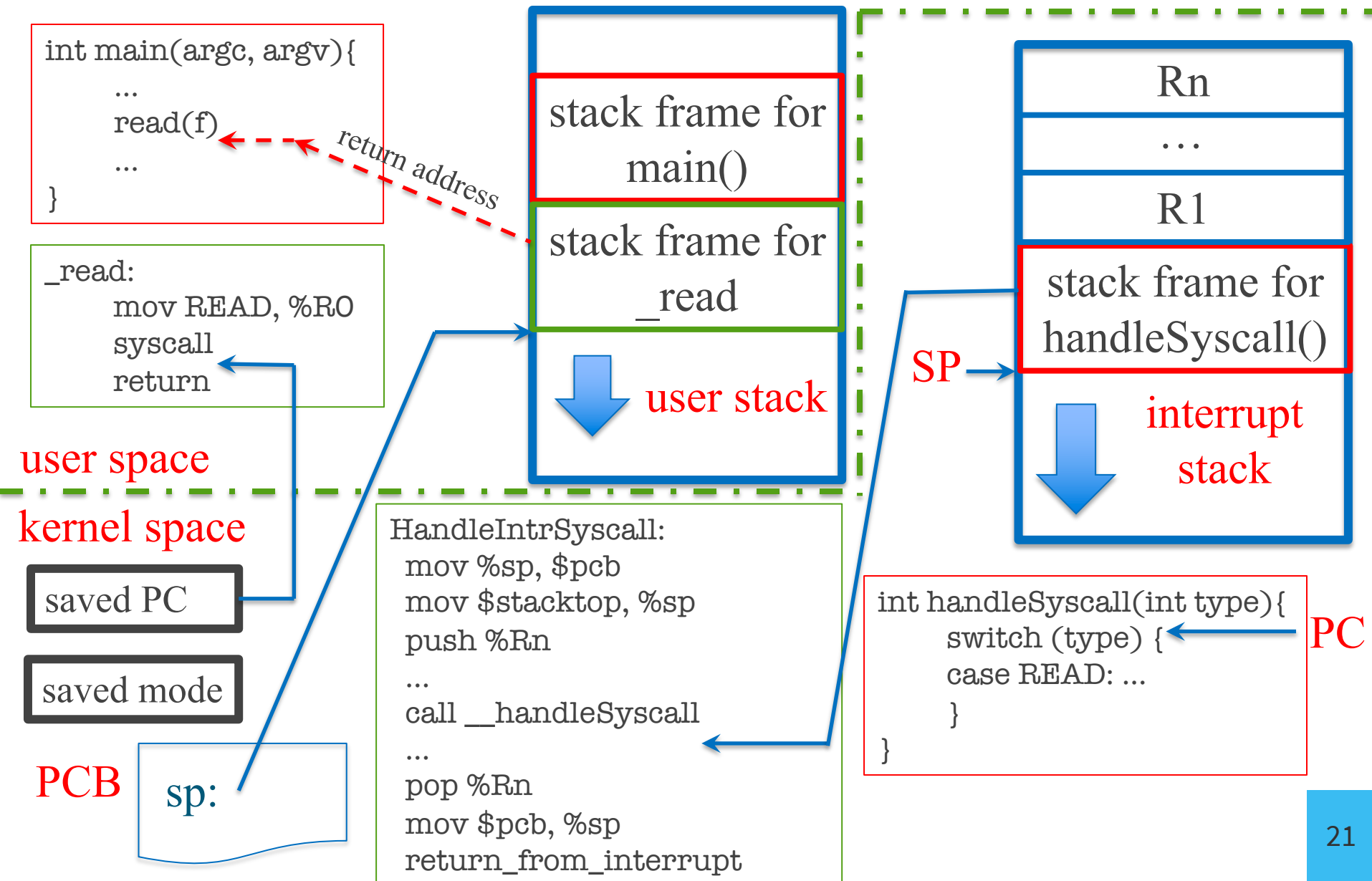
# Executing read System Call



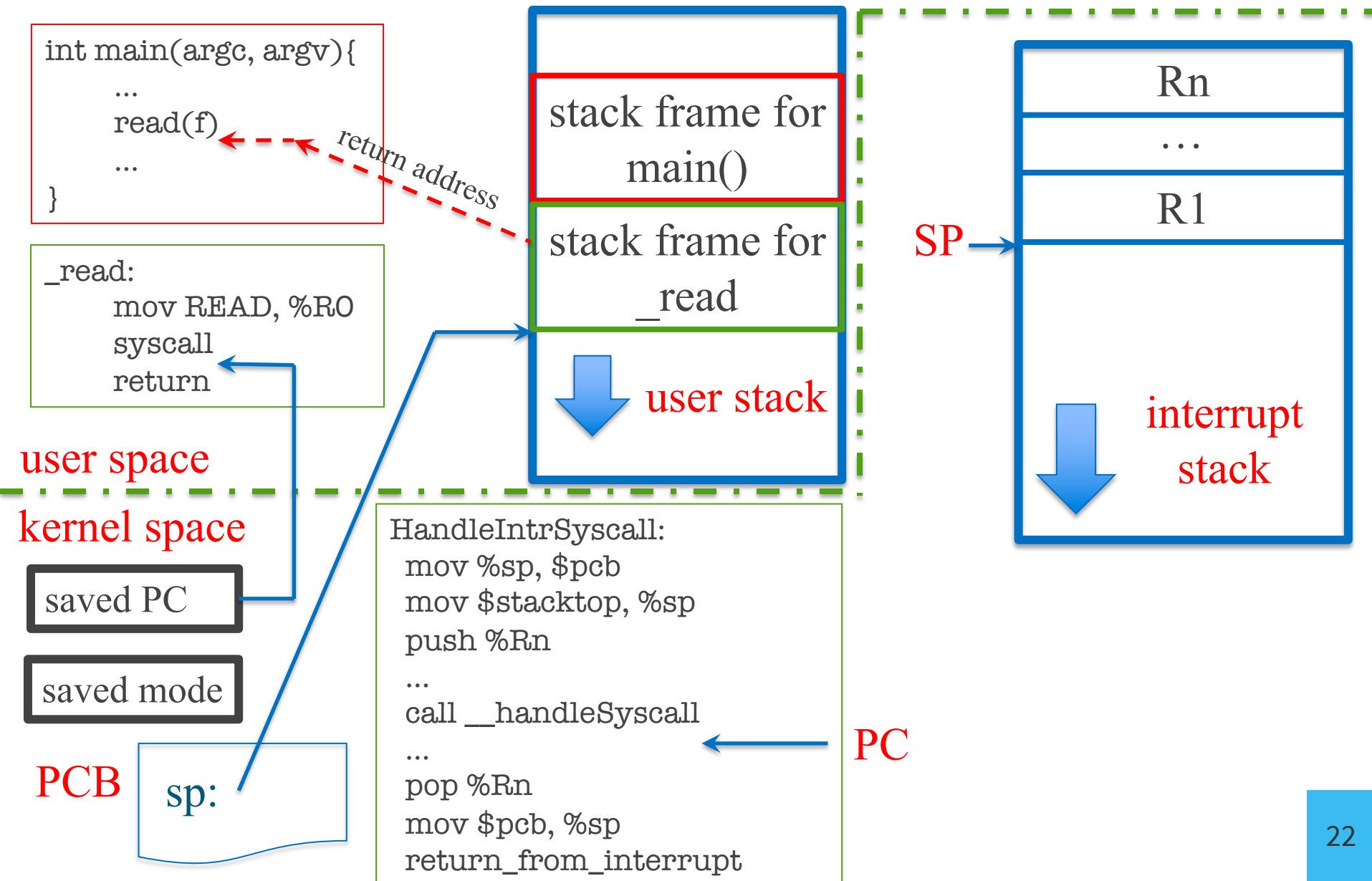
# Executing read System Call



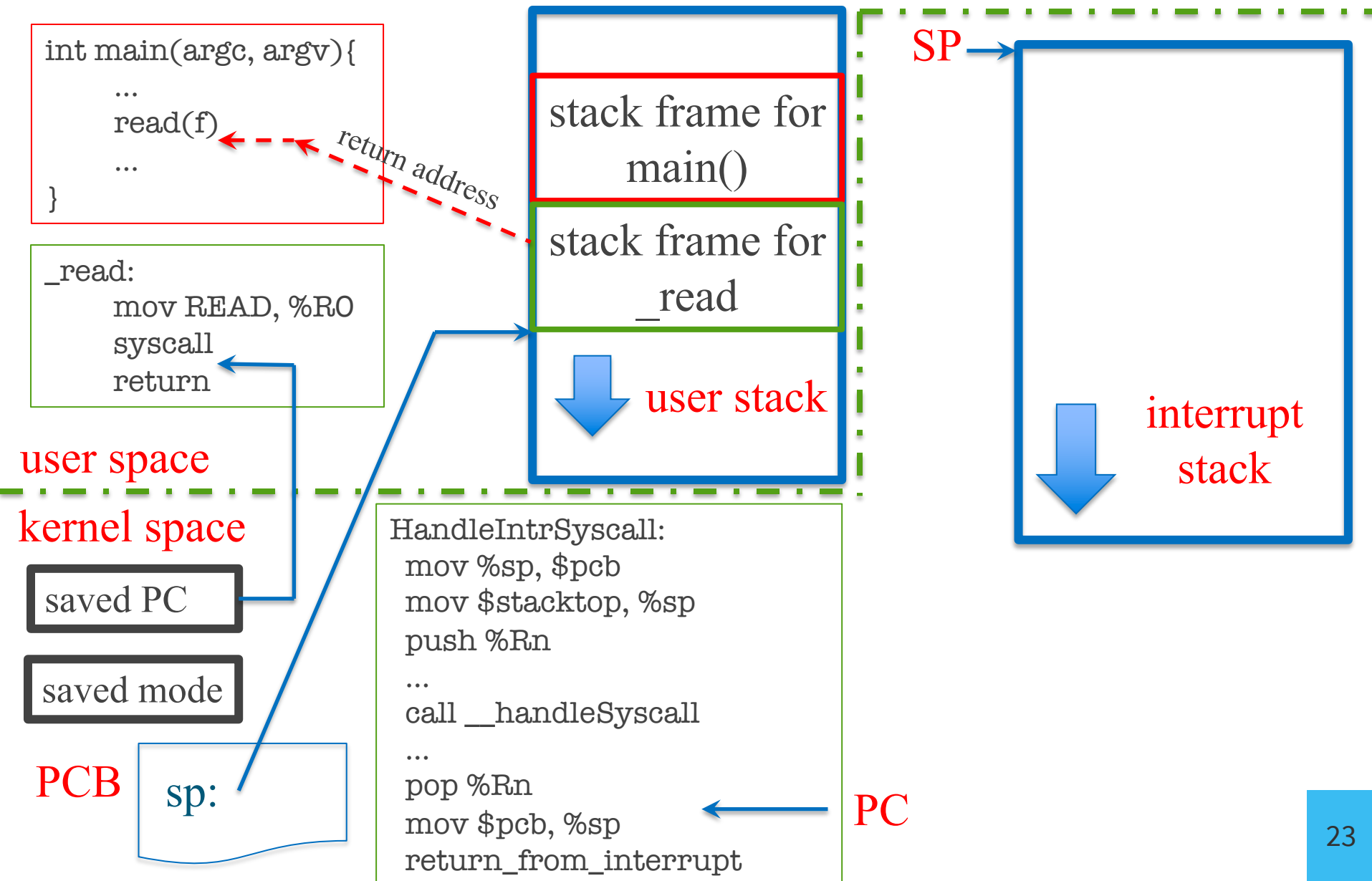
# Executing read System Call



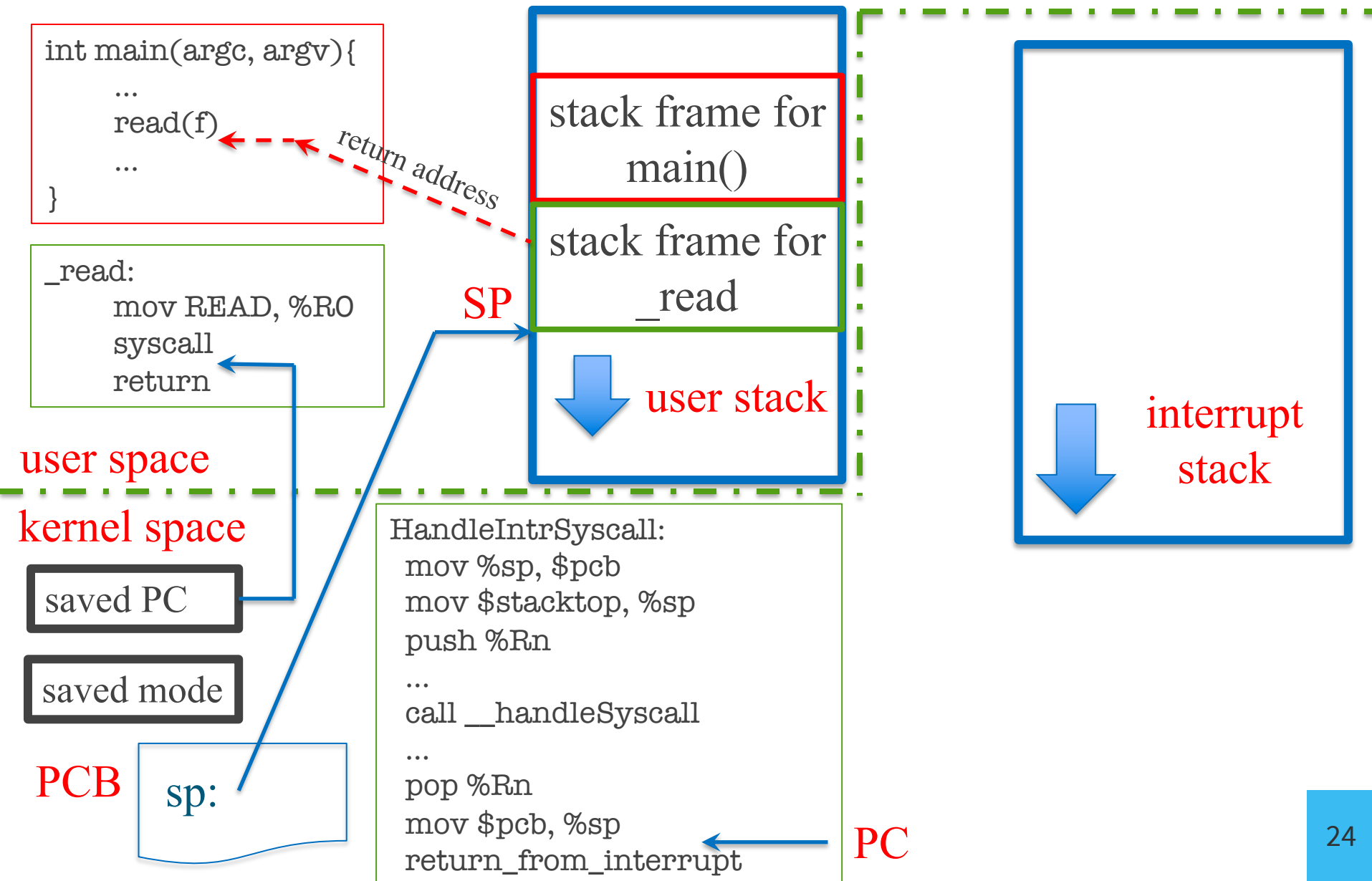
# Executing read System Call



# Executing read System Call

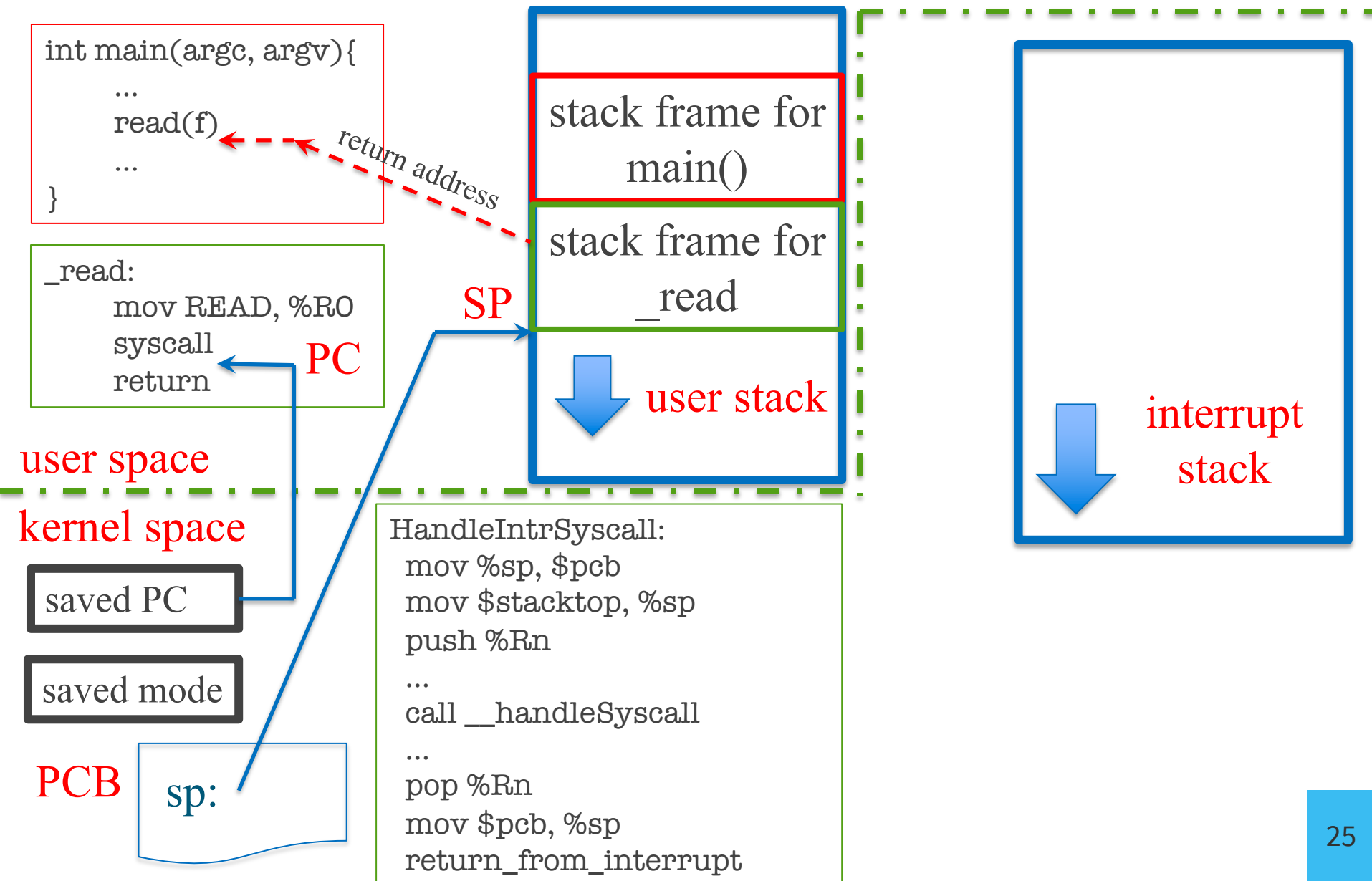


# Executing read System Call





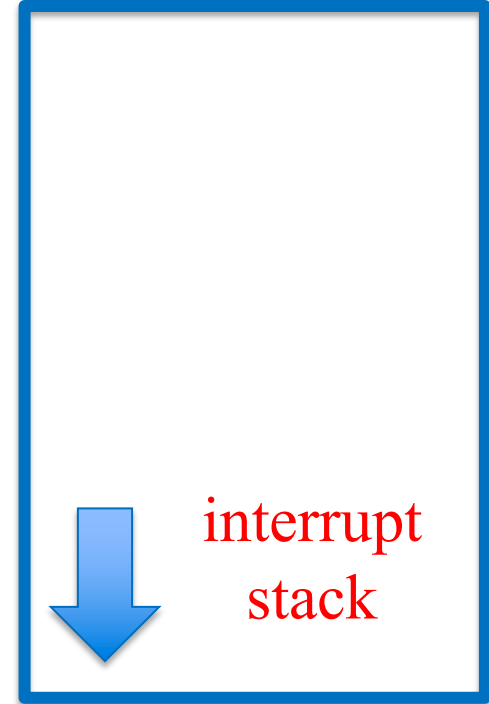
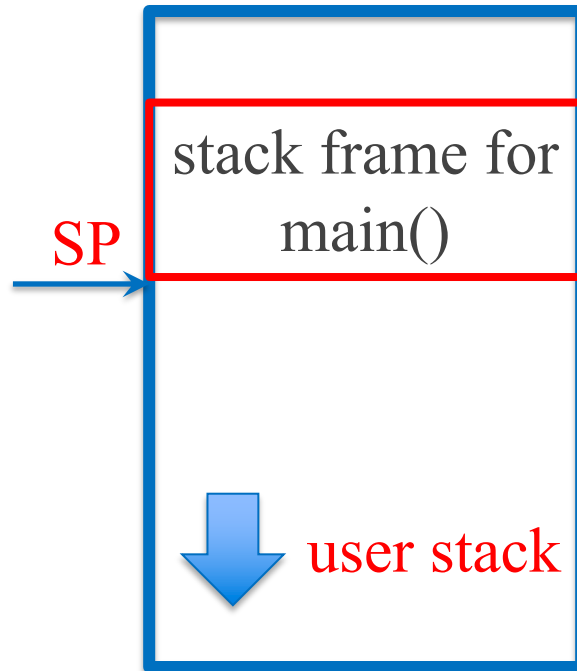
# Executing read System Call



# Executing read System Call

```
int main(argc, argv){  
    ...  
    read(f) ← PC  
    ...  
}
```

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



user space

kernel space

# What if read needs to “block”?

- read may need to block if
  - reading from terminal
  - reading from disk and block not in cache
  - reading from remote file server

should run another process!

# How to run multiple processes?

*(on a single core)*

# A process physically runs on the CPU

But *somehow* each process has its own:

- ◆ Registers
  - ◆ Memory
  - ◆ I/O resources
  - ◆ “thread of control”
- *even though there are usually more processes than the CPU has cores*
    - ➔ *need to multiplex, schedule, ... to create virtual CPUs for each process*

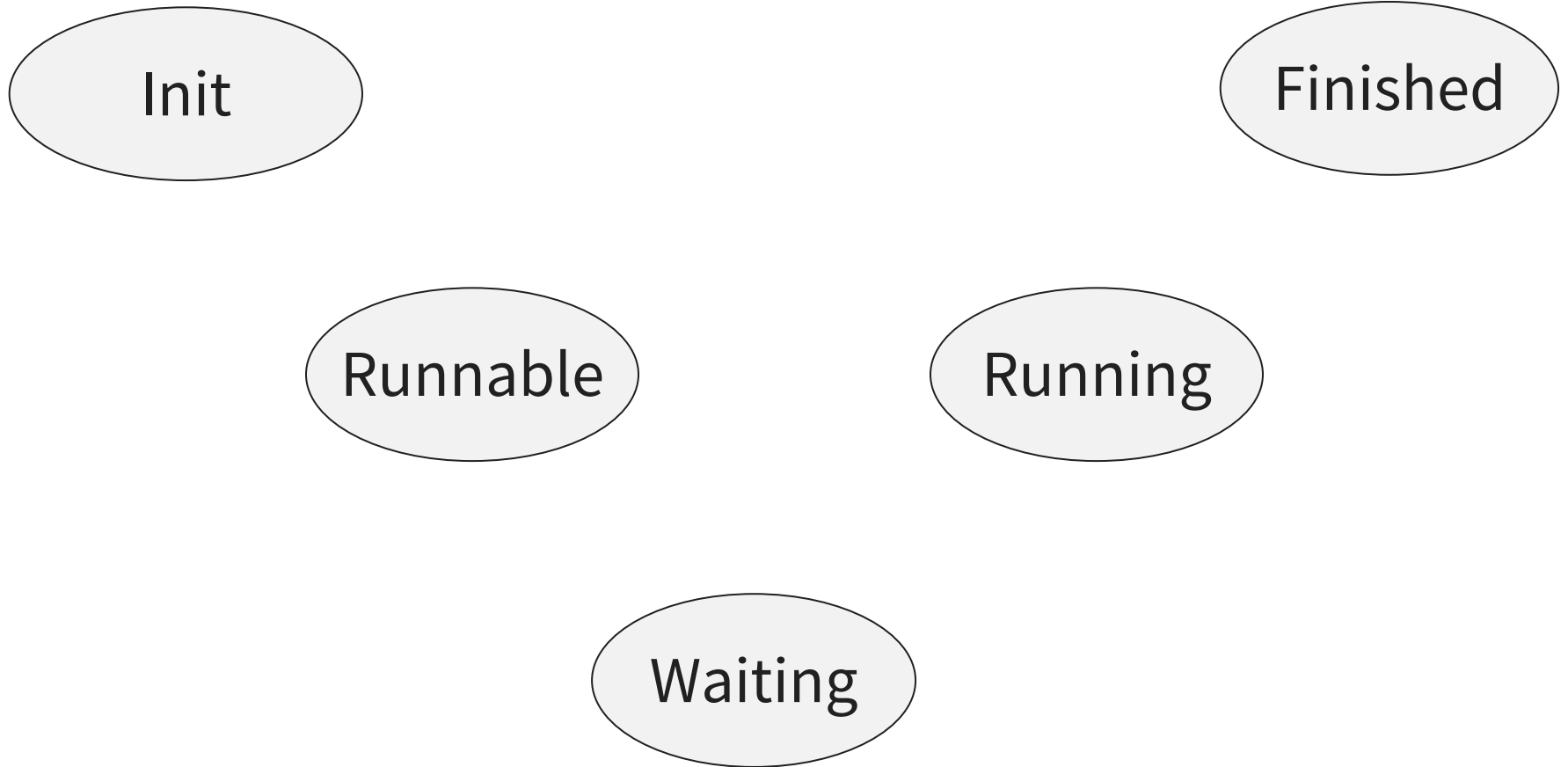
*For now, assume we have a single core CPU*

# Process Control Block (PCB)

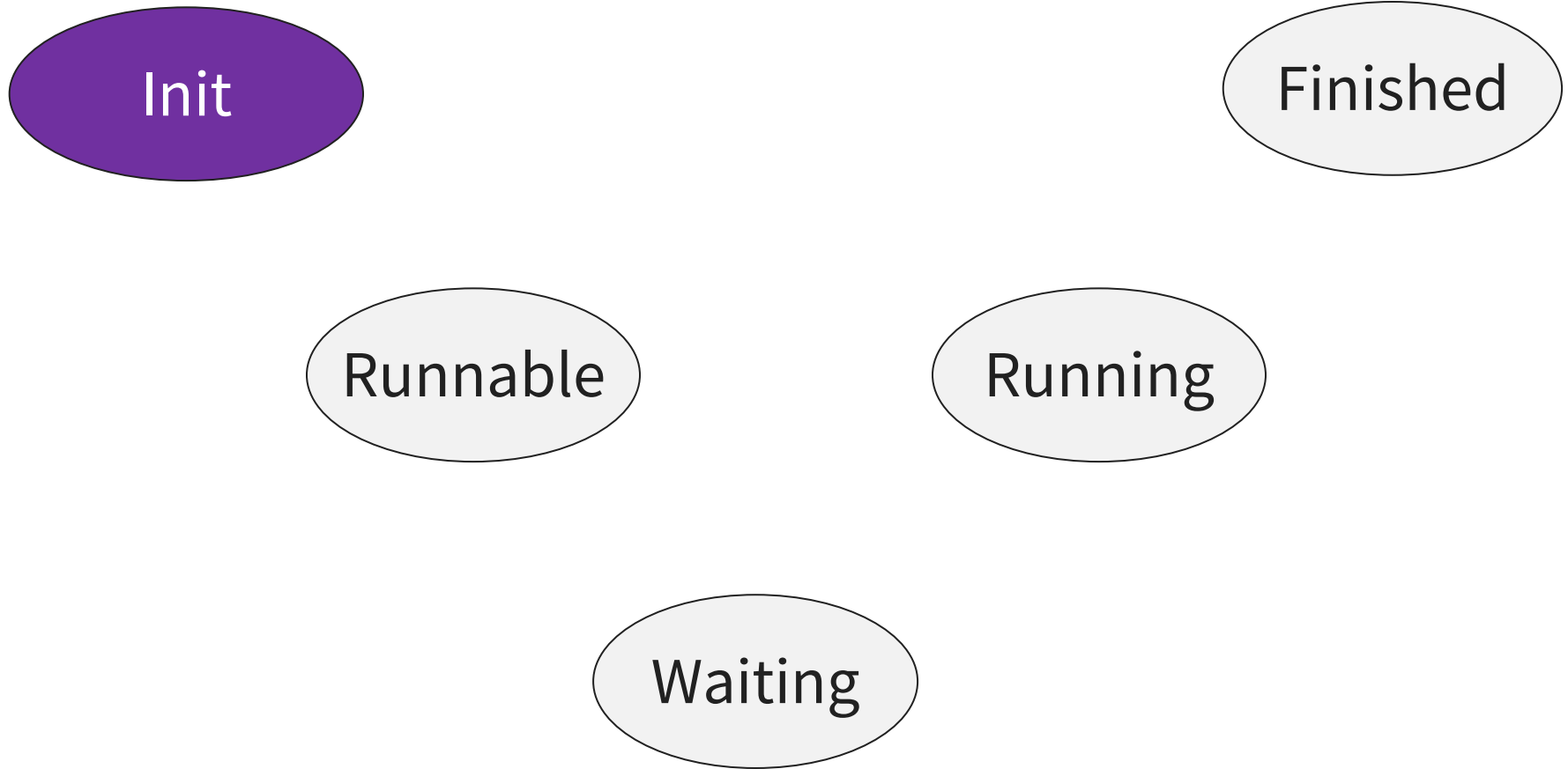
For each process, the OS has a PCB containing:

- location in memory (page table)
- location of executable on disk
- which user is executing this process (`uid`)
- process identifier (`pid`)
- process status (running, waiting, finished, etc.)
- scheduling information
- interrupt stack
- saved user SP
  - points into user stack
- saved kernel SP
  - points into interrupt stack
  - interrupt stack contains saved registers and kernel call stack for this process
- ... *and more!*

# Process Life Cycle



# Process creation

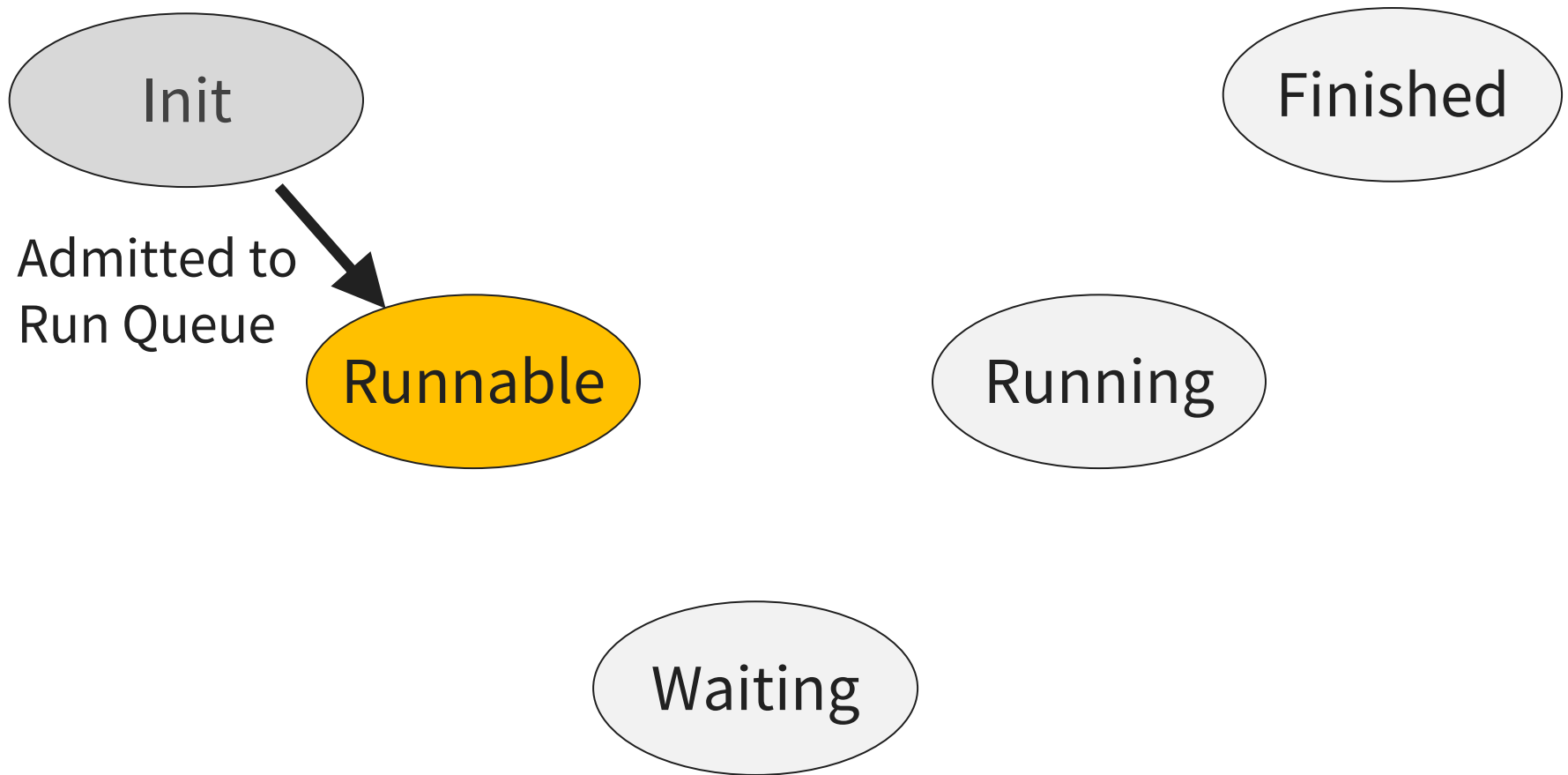


**PCB status:** being created

**Registers:** uninitialized



# Process is Ready to Run

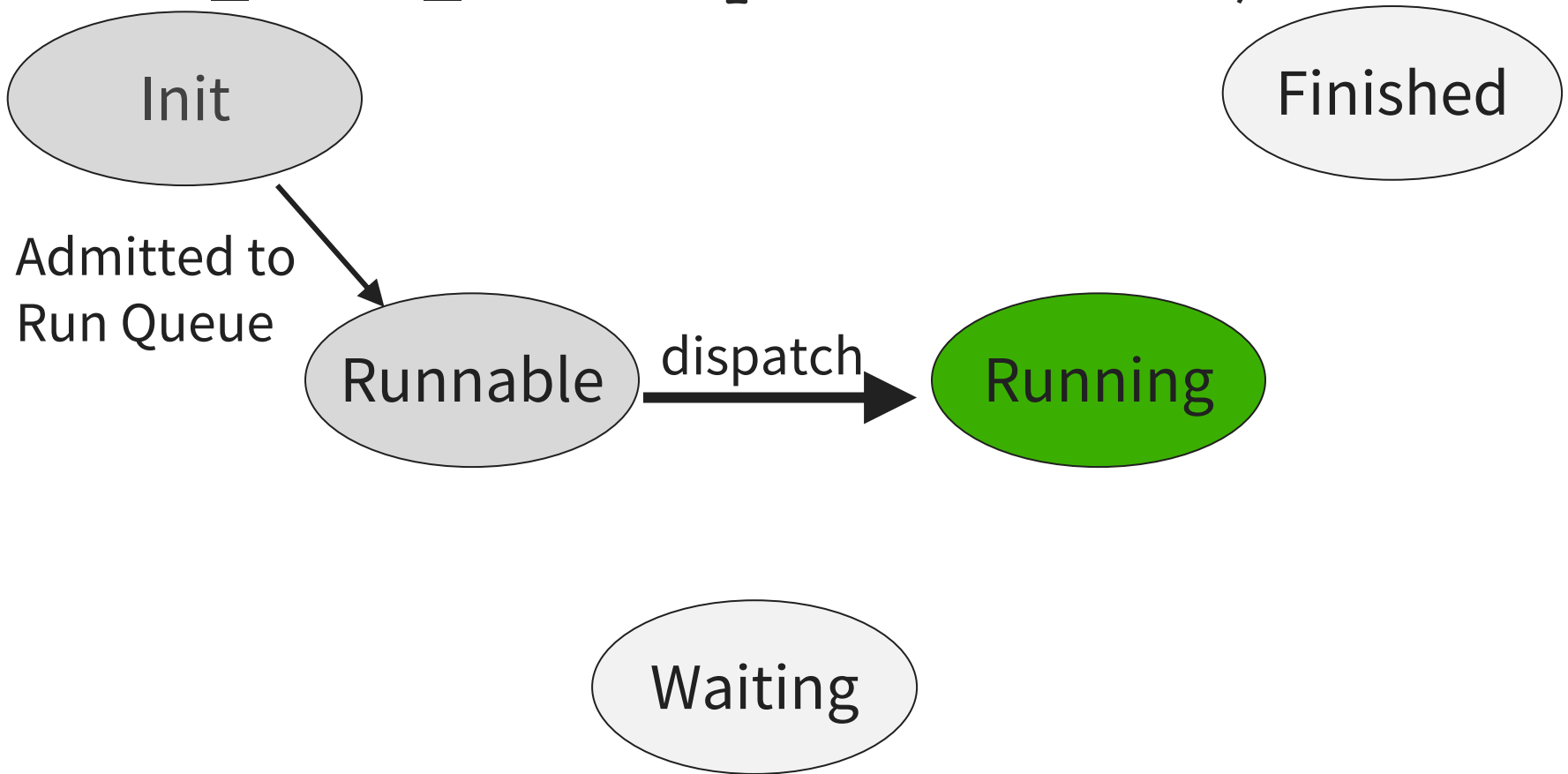


**PCB:** on Run Queue (aka Ready Queue)

**Registers:** pushed by kernel code onto interrupt stack

# Process is Running

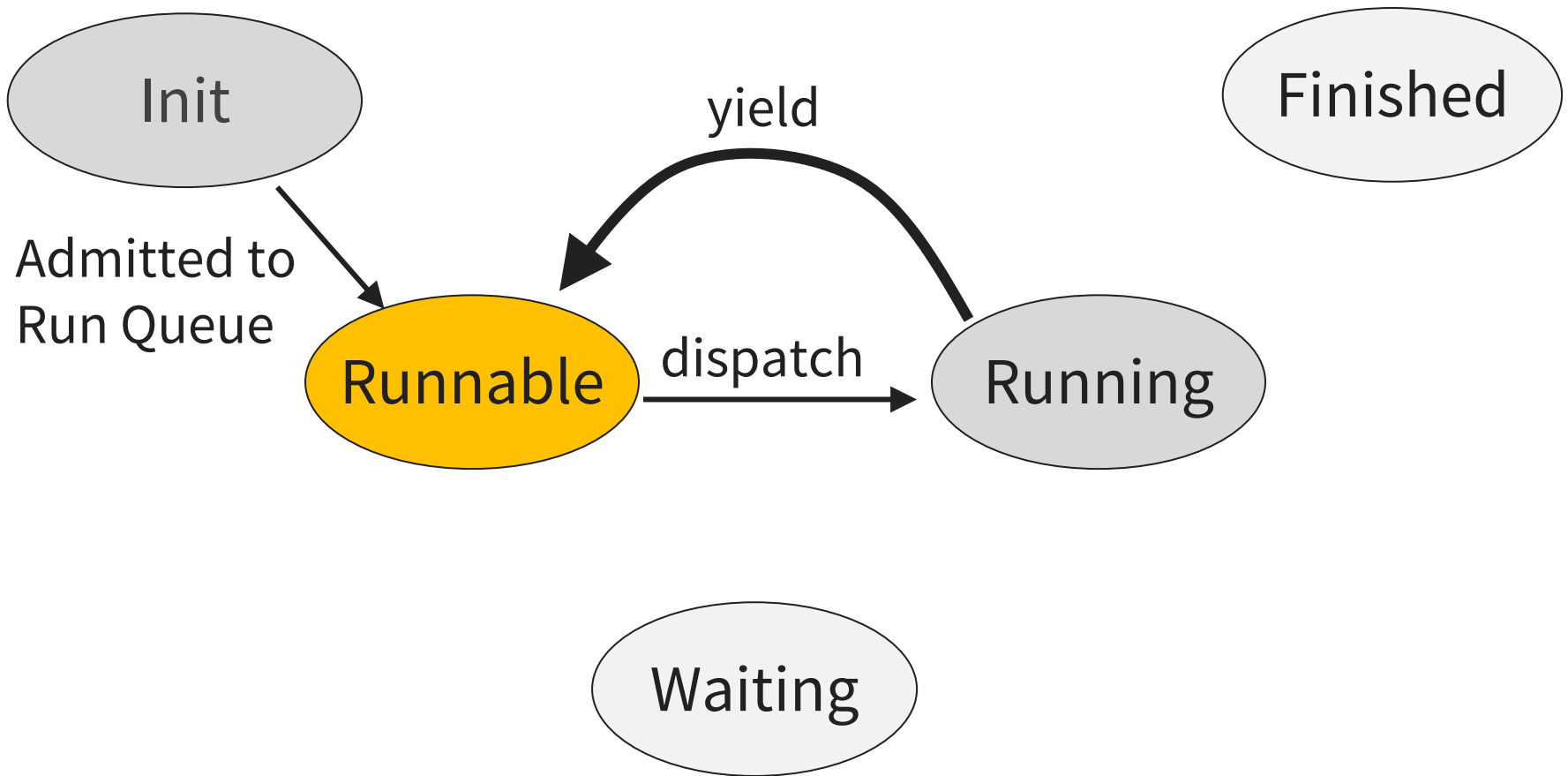
(in supervisor mode, but may  
return\_from\_interrupt to user mode)



**PCB:** currently executing

**Registers:** popped from interrupt stack into CPU

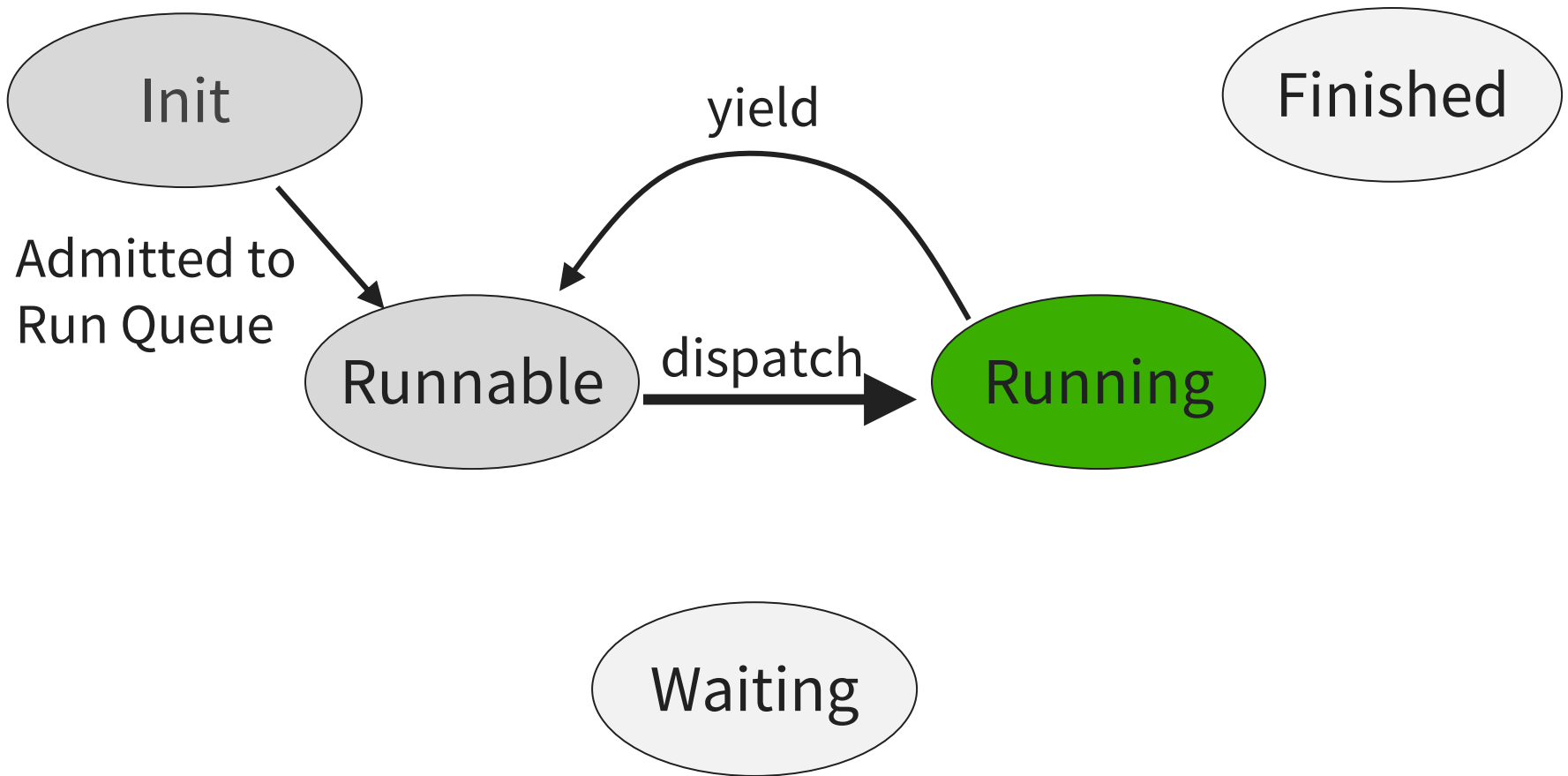
# Process Yields (on clock interrupt)



**PCB:** on Run queue

**Registers:** pushed onto interrupt stack (sp saved in PCB)

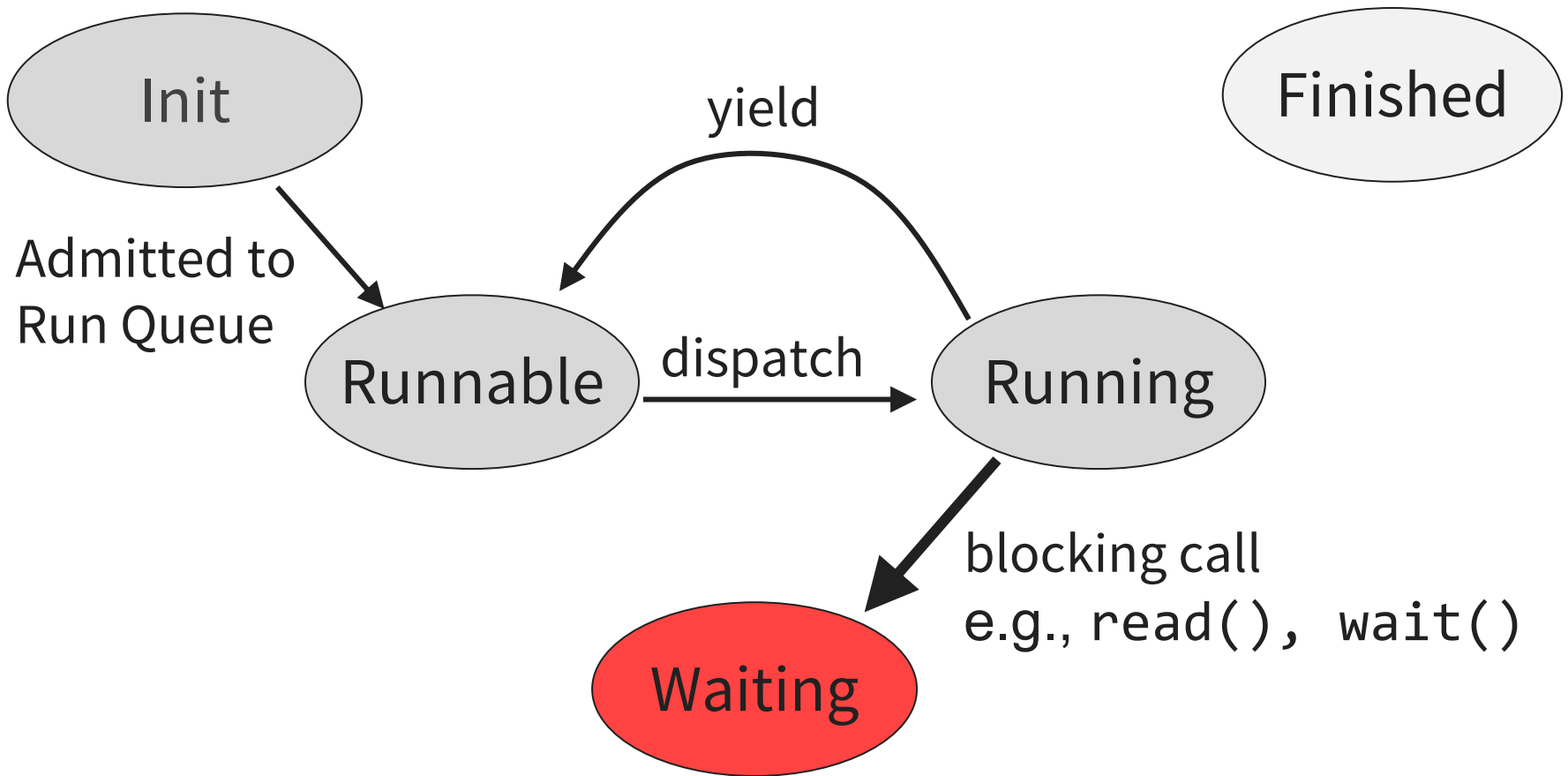
# Process is Running Again!



**PCB:** currently executing

**Registers:** sp restored from PCB; others restored from stack

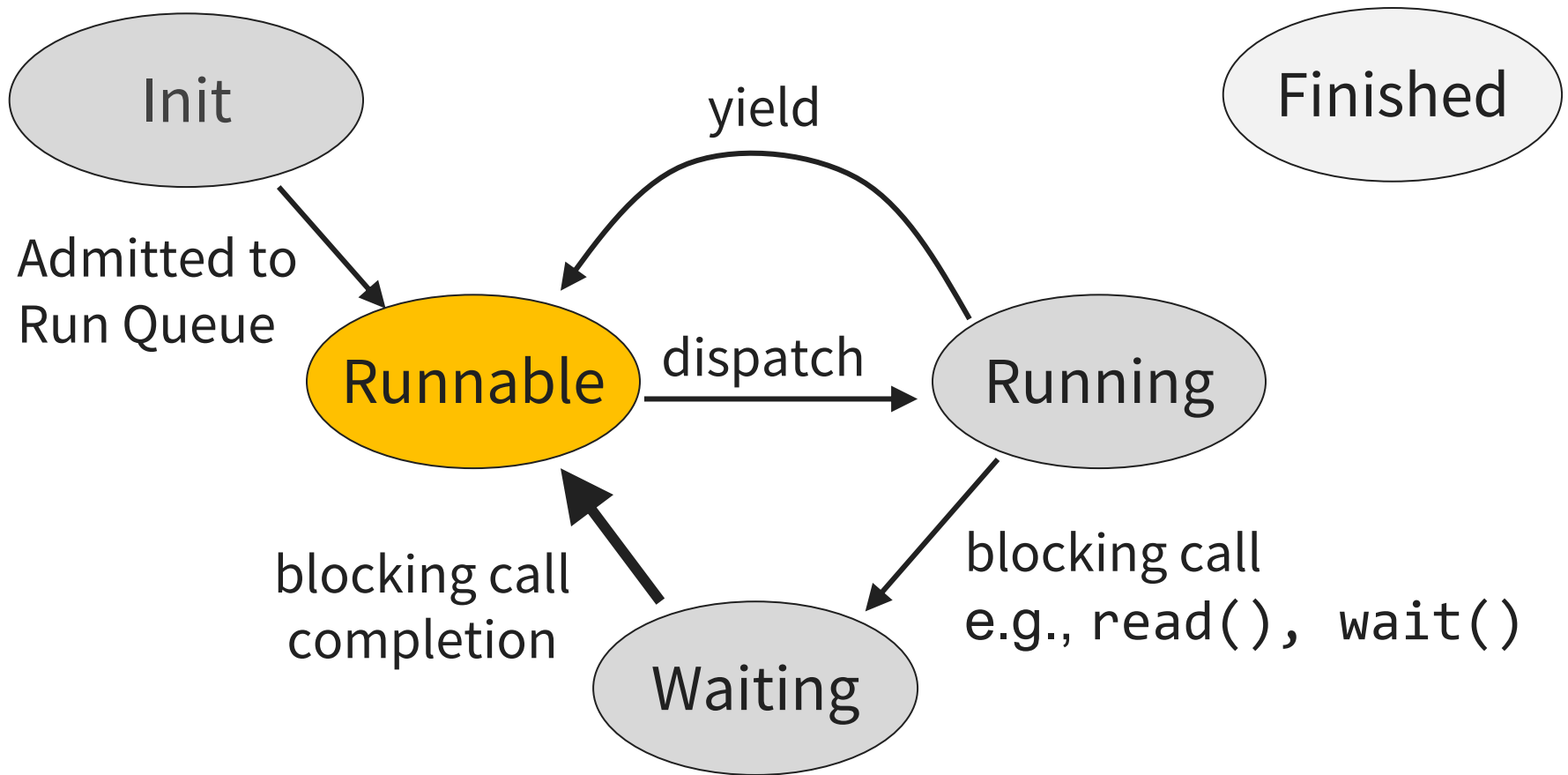
# Process is Waiting



**PCB:** on specific waiting queue (file input, ...)

**Registers:** on interrupt stack

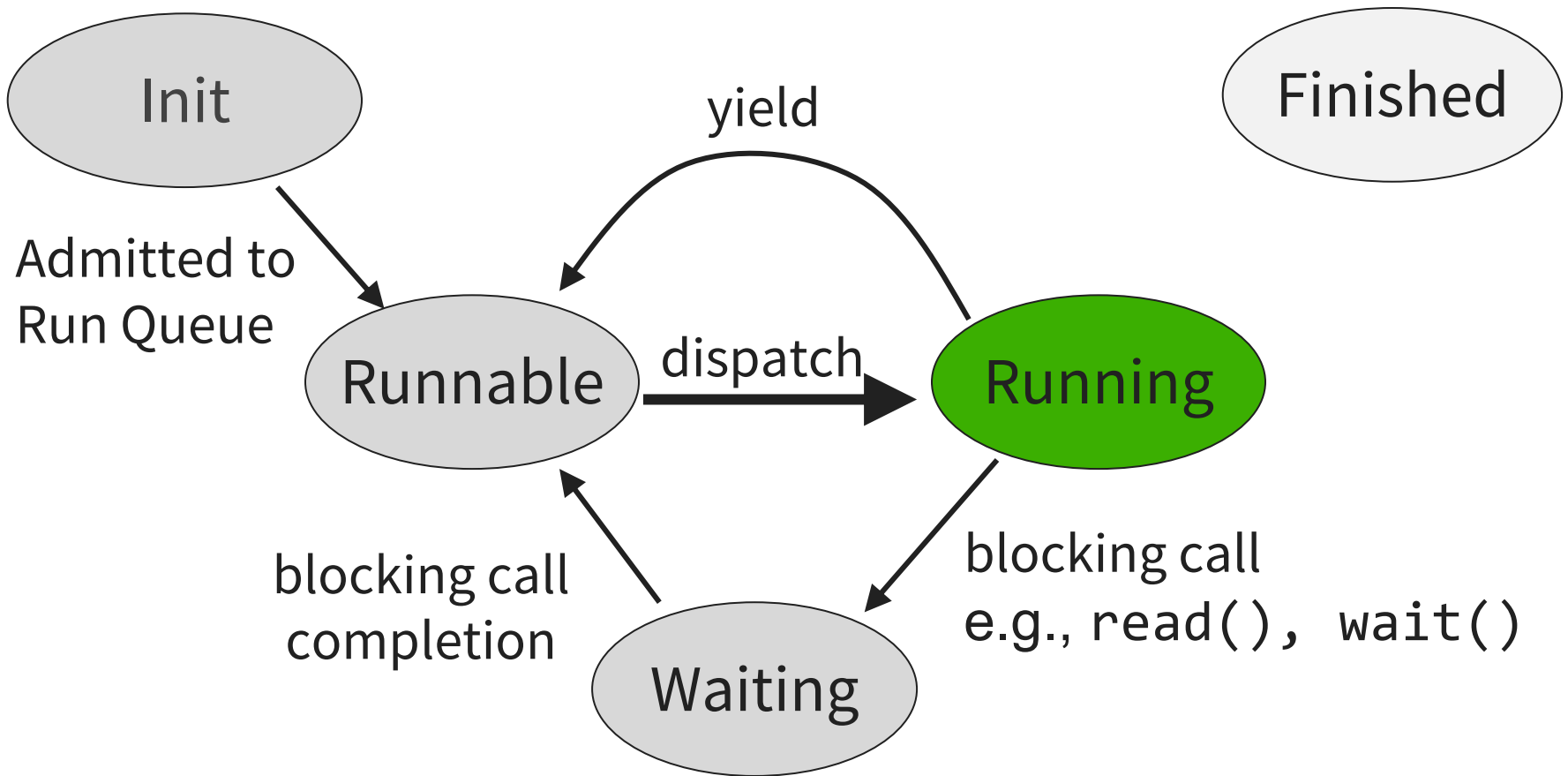
# Process is Ready Again!



**PCB:** on run queue

**Registers:** on interrupt stack

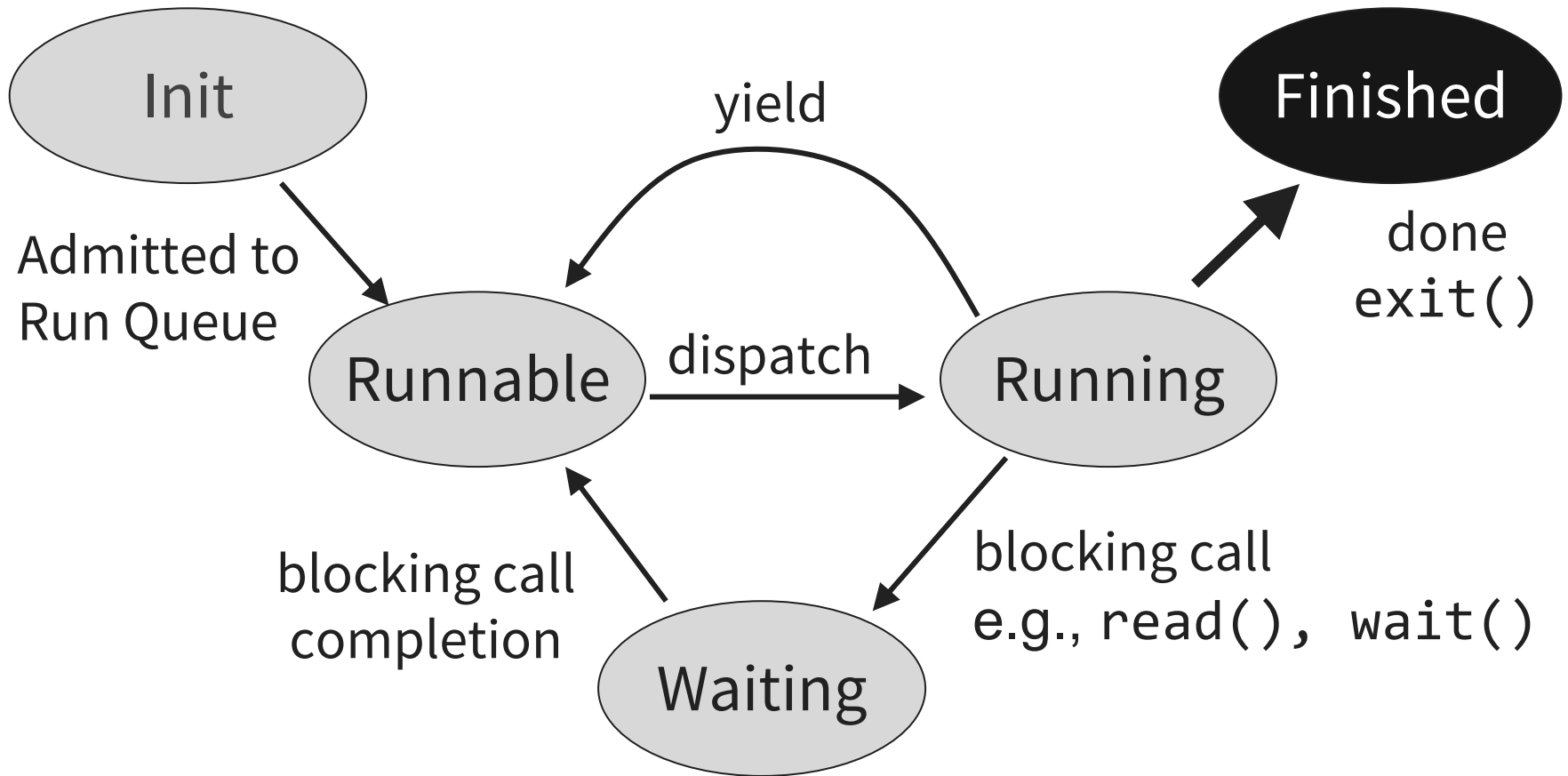
# Process is Running Again!



**PCB:** currently executing

**Registers:** restored from interrupt stack into CPU

# Process is Finished (Process = Zombie)



**PCB:** on Finished queue, ultimately deleted  
**Registers:** no longer needed



# Invariants to keep in mind

- At most 1 process is RUNNING at any time (*per core*)
- When CPU is in user mode, current process is RUNNING and its interrupt stack is empty
- If process is RUNNING
  - its PCB is not on any queue
  - *however, not necessarily in user mode*
- If process is RUNNABLE or WAITING
  - its interrupt stack is non-empty and can be switched to
    - i.e., has its registers saved on top of the stack
  - its PCB is either
    - on the run queue (if RUNNABLE)
    - on some wait queue (if WAITING)
- If process is FINISHED
  - its PCB is on finished queue

# Cleaning up zombies

- Process cannot clean up itself

## WHY NOT?

- Process can be cleaned up
  - either by any other process
    - check for zombies just before returning to RUNNING state
  - or by parent when it waits for it
    - but what if the parent dies first?
  - or by dedicated “reaper” process
- Linux uses combination:
  - usually parent cleans up child process when waiting
  - if parent dies before child, child process is inherited by the initial process, which is continually waiting



# How To Yield/Wait?

*Switching from executing the current process to another runnable process*

- Process 1 goes from RUNNING → RUNNABLE/WAITING
  - Process 2 goes from RUNNABLE → RUNNING
1. save kernel registers of process 1 on its interrupt stack
  2. save kernel sp of process 1 in its PCB
  3. restore kernel sp of process 2 from its PCB
  4. restore kernel registers from its interrupt stack

# ctx\_switch(&old\_sp, new\_sp)

```
ctx_switch:
    addi sp,sp,-64 // reserve frame
    sw s0,4(sp)
    sw s1,8(sp)
    sw s2,12(sp)
    sw s3,16(sp)
    sw s4,20(sp)
    sw s5,24(sp)
    sw s6,28(sp)
    sw s7,32(sp)
    sw s8,36(sp)
    sw s9,40(sp)
    sw s10,44(sp)
    sw s11,48(sp)
    sw ra,52(sp) // save return addr
    sw sp,0(a0) // save old sp
    mv sp,a1 // set new sp
    lw s0,4(sp)
    lw s1,8(sp)
    lw s2,12(sp)
    lw s3,16(sp)
    lw s4,20(sp)
    lw s5,24(sp)
    lw s6,28(sp)
    lw s7,32(sp)
    lw s8,36(sp)
    lw s9,40(sp)
    lw s10,44(sp)
    lw s11,48(sp)
    lw ra,52(sp) // return addr
    addi sp,sp,64 // free frame
    ret // return
```

(author: Yunhao Zhang)

## USAGE:

```
struct pcb *current, *next;
```

```
void yield(){
    assert(current->state == RUNNING);
    current->state = RUNNABLE;
    runQueue.add(current);
    next = scheduler();
    next->state = RUNNING;
    ctx_switch(&current->sp, next->sp)
    current = next;
}
```

# What if there are no more RUNNABLE processes?

- `scheduler()` would return `NULL` and things blow up
- solution: always run a low priority process that sits in an infinite loop executing the RISC-V WFI (Wait For Interrupt) or x86 HLT instruction or ... (fill in your favorite CPU)
  - which waits for the next interrupt, saving energy when there's nothing to do

# Three “kinds” of context switches

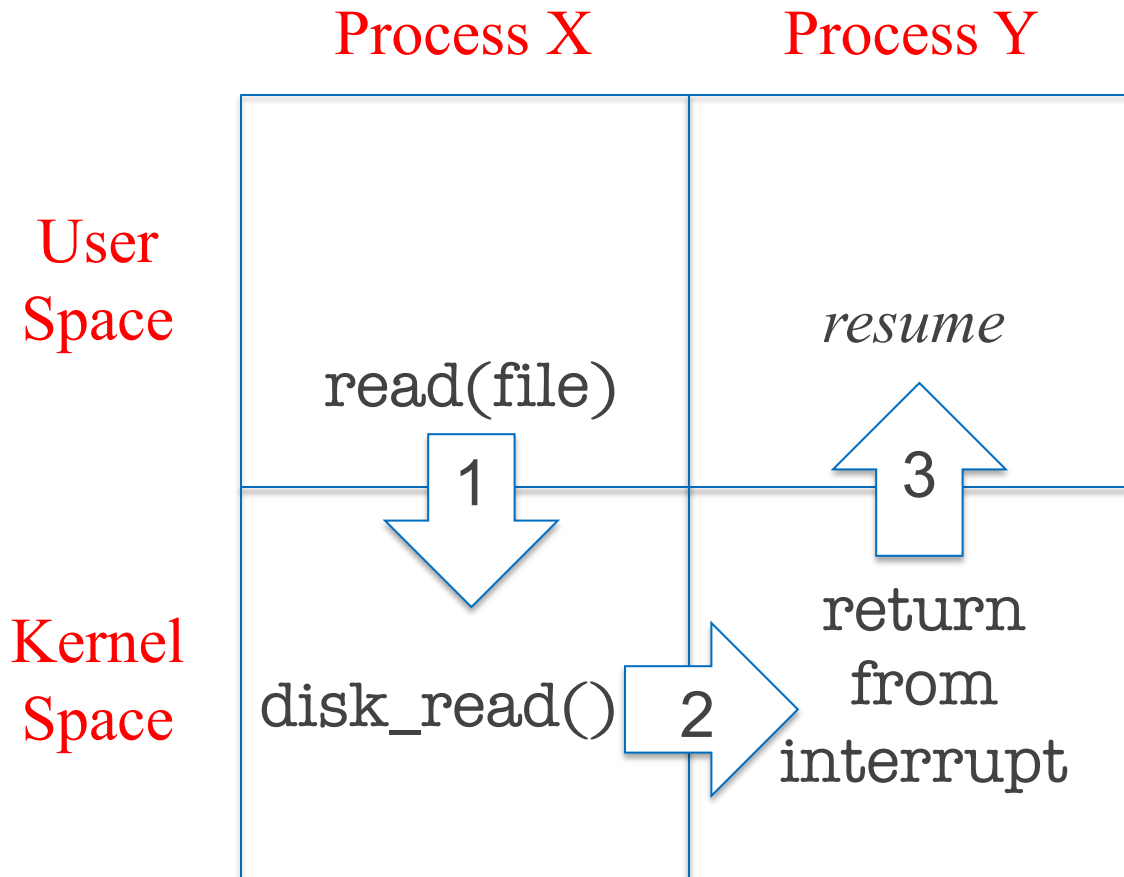
1. **Interrupt**: From user to kernel space
  - system call, exception, or interrupt
2. **Yield**: In kernel space, between two processes
  - happens inside the kernel, switching from one PCB/interrupt stack to another
3. **Return-From-Interrupt**: From kernel space to user space
  - Through a `return_from_interrupt` instruction

Note that each involves a stack switch:

1. Px user stack → Px interrupt stack
2. Px interrupt stack → Py interrupt stack
3. Py interrupt stack → Py user stack

A **context** is “the CPU state,” which is captured in its registers. By context switching, the CPU can play different roles at different times

# Example switch between processes



1. save process X user registers
2. save process X kernel registers and restore process Y kernel registers
3. restore process Y user registers

*before step 2: scheduler picks a runnable process*

# A word on “abstraction”

- We manage complexity through abstraction
- When I say “tea water,” I mean the water that is used for tea
  - but it’s just water
  - that same water will serve different purposes in its existence
- When I say “kernel memory,” I mean the memory that is used for the kernel
  - but it’s just memory
  - it’s the same kind of memory that is used for processes
- Actors in a play: same actors can play multiple roles in their lives, sometimes even in the same play
  - actors are time multiplexed, same as registers of a CPU
  - the kernel SP is just the SP that is used by the kernel
  - when you’re watching “Woman King,” you’re supposed to imagine seeing *Nanisca*, not *Viola Davis*



# A “process” is an abstraction

- Abstract computer with abstract memory, registers, and peripherals
- Some “hardware” computer can be multiplexed to run multiple processes
  - *time multiplexed*: registers
  - *space multiplexed*: disk

# Review

- A *process* is an abstraction of a computer
- A *context* captures the state of the processor:
  - registers (including PC and SP)
- The *implementation* uses *two* contexts:
  - user context
  - kernel (supervisor) context
- A *Process Control Block (PCB)* is a kernel data structure that saves contexts and has other information about the process

# System calls to create a new process

Windows:

`CreateProcess(...);`

UNIX (Linux):

`fork() + exec(...)`

# CreateProcess (Simplified)

## System Call:

```
if (!CreateProcess(  
    NULL,    // No module name (use command line)  
    argv[1], // Command line  
    NULL,    // Process handle not inheritable  
    NULL,    // Thread handle not inheritable  
    FALSE,   // Set handle inheritance to FALSE  
    0,       // No creation flags  
    NULL,    // Use parent's environment block  
    NULL,    // Use parent's starting directory  
    &si,      // Pointer to STARTUPINFO structure  
    &pi )    // Ptr to PROCESS_INFORMATION structure  
)
```

# ~~CreateProcess~~ (Simplified) ~~fork~~ (actual form)

System Call:

```
int pid = fork( void 😊  
    NULL, // No module name (use command line)  
    argv[1], // Command line  
    NULL, // Process handle not inheritable  
    NULL, // Thread handle not inheritable  
    FALSE, // Set handle inheritance to FALSE  
    0, // No creation flags  
    NULL, // Use parent's environment block  
    NULL, // Use parent's starting directory  
    &si, // Pointer to STARTUPINFO structure  
    &pi )  
)
```

**pid** = process identifier

# Kernel actions to create a process

## **fork():**

- Allocate ProcessID
- Create & initialize PCB
- Create and initialize a new address space
- Inform scheduler that new process is ready to run

## **exec(program, arguments):**

- Load the program into the address space
- Copy arguments into memory in address space
- Initialize h/w context to start execution at “start”

Windows **createProcess(...)** does both

# Creating and Managing Processes

<b>fork()</b>	Create a child process as a clone of the current process. <b>Returns to both parent and child</b> . Returns child pid to parent process, 0 to child process.
<b>exec</b> ( <b>prog</b> , args)	Run the application <b>prog</b> in the current process with the specified arguments ( <i>replacing any code and data that was in the process already</i> )
<b>wait</b> (&status)	Pause until a child process has exited
<b>exit</b> (status)	Tell the kernel the current process is complete and should be garbage collected.
<b>kill</b> (pid, type)	Send an interrupt of a specified type to a process. (a bit of a misnomer, no?)

# Fork + Exec

Process 1

Program A

PC → `child_pid = fork();`  
`if (child_pid==0)`  
    `exec(B);`  
`else`  
    `wait(&status);`

`child_pid` ?



# Fork + Exec

*fork returns  
twice!*

Process 1  
Program A

```
child_pid = fork();  
PC → if (child_pid == 0)  
      exec(B);  
      else  
        wait(&status);
```

child\_pid 42

Process 42  
Program A

```
child_pid = fork();  
PC → if (child_pid == 0)  
      exec(B);  
      else  
        wait(&status);
```

child\_pid 0

# Fork + Exec

Process 1  
Program A

```
child_pid = fork();  
if (child_pid==0)  
    exec(B);  
else
```

PC → `wait(&status);`

child\_pid 42



*Waits until child exits.*

Process 42  
Program A

```
child_pid = fork();  
if (child_pid==0)  
    exec(B);  
else  
    wait(&status);
```

PC →

child\_pid 0

# Fork + Exec

Process 1  
Program A

```
child_pid = fork();  
if (child_pid==0)  
    exec(B);  
else
```

PC → wait(&status);

child\_pid 42

Process 42  
Program A

```
child_pid = fork();  
if (child_pid==0)  
    exec(B);  
else  
    wait(&status);
```

PC →

child\_pid 0



***if and else  
both executed!***

# Fork + Exec

Process 1  
Program A

```
child_pid = fork();  
if (child_pid==0)  
    exec(B);  
else  
    wait(&status);
```

PC →

child\_pid 42



Process 42  
Program B

PC →

```
main() {  
    ...  
    exit(3);  
}
```

# Fork + Exec

Process 1

Program A

```
child_pid = fork();  
if (child_pid==0)  
    exec(B);  
else
```

PC → **wait(&status);**

child\_pid 42

status 3



# Code example (fork.c)

```
#include <stdio.h>
#include <unistd.h>
```

```
int main() {
    int child_pid = fork();

    if (child_pid == 0) {        // child process
        printf("I am process %d\n", getpid());
    }
    else {                      // parent process.
        printf("I am the parent of process %d\n", child_pid);
    }
    return 0;
}
```

Possible outputs?

# Threads! (Chapters 25-27)

Other terms for threads:

- Lightweight Process
- Thread of Control
- Task

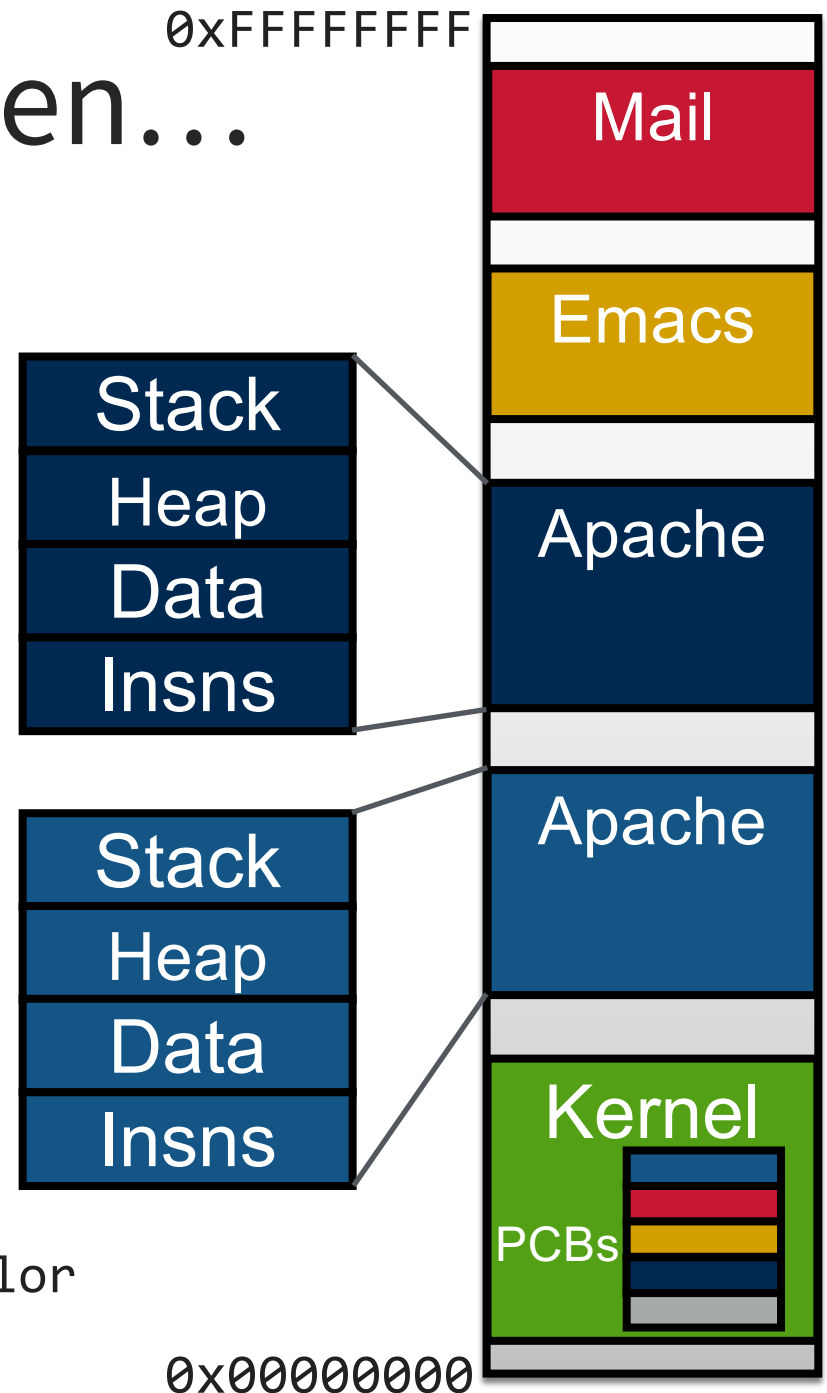
# What happens when...

Apache wants to run multiple concurrent computations?

Two heavyweight address spaces for two concurrent computations

Hard to share cache, etc.

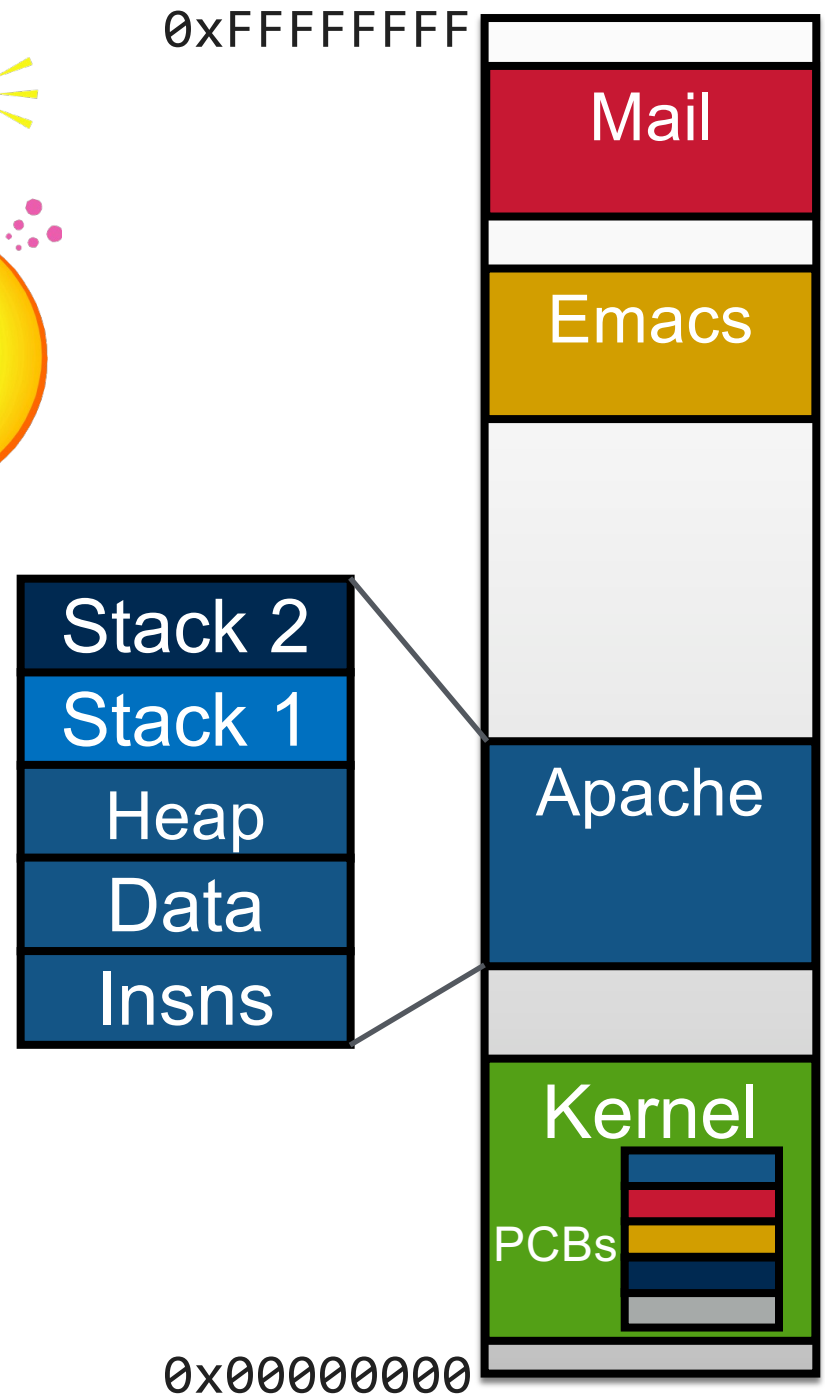
Physical address space  
Each process' address space by color  
(shown contiguous to look nicer)





# Idea

Place concurrent computations in the same address space!



# Process vs. Thread Abstraction

- A process is an abstraction of a computer
  - CPU, memory, devices
- A thread is an abstraction of a core
  - registers (incl. PC and SP)

*Unbounded #computers, each with unbounded #cores*

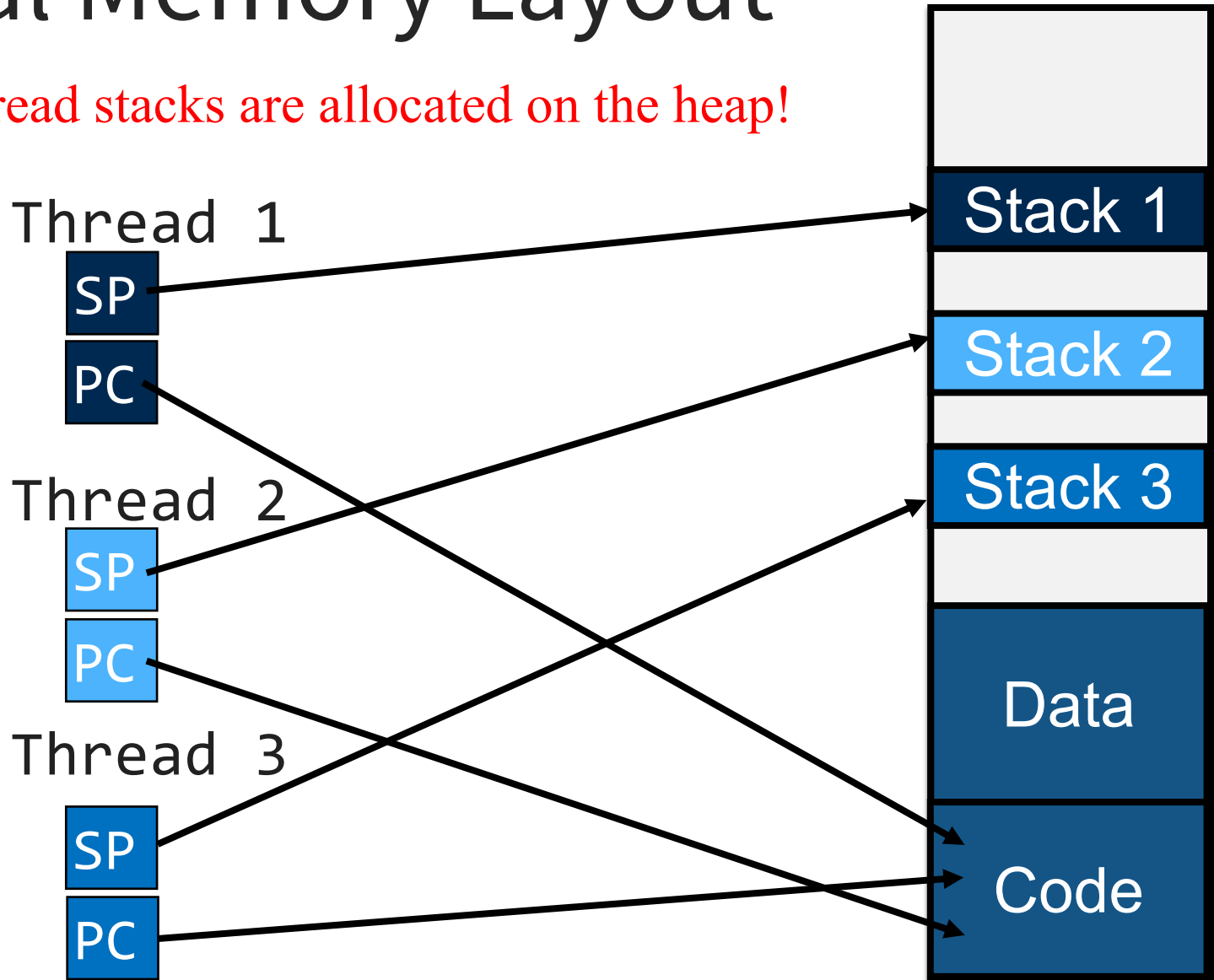
- Different processes typically have their own (virtual) memory, but different threads share virtual memory.
- Different processes tend to be mutually distrusting, but threads must be mutually trusting. *Why?*

# *Nomenclature Warning*

- In “concurrency literature”, threads are often called “processes” or “processors”

# Virtual Memory Layout

Thread stacks are allocated on the heap!



# Why Threads?



## Concurrency

- exploiting multiple CPUs/cores

## Mask long latency of I/O

- doing useful work while waiting

## Responsiveness

- high priority GUI threads / low priority work threads

## Encourages natural program structure

- Expressing logically concurrent tasks
- update screen, fetching data, receive user input

# Some Thread Examples

```
for (k = 0; k < n; k++) {  
    a[k] = b[k] × c[k] + d[k] × e[k]  
}
```

Web server:

1. get network message (URL) from client
2. get URL data from disk
3. compose response
4. send response

# Simple Thread API

<code>void thread_create (<b>func</b>, <b>arg</b>)</code>	Creates a new thread that will execute function <b>func</b> with the arguments <b>arg</b>
<code>void thread_yield()</code>	Calling thread gives up processor. Scheduler can resume running this thread at any point.
<code>void thread_exit()</code>	Finish caller

# Preemption

- Two kinds of threads:
  - **Non-preemptive**: explicitly yield to other threads
  - **Preemptive**: yield automatically upon clock interrupts
- Most modern threading systems are preemptive
  - but not 4411 P1 project



# Implementation of Threads

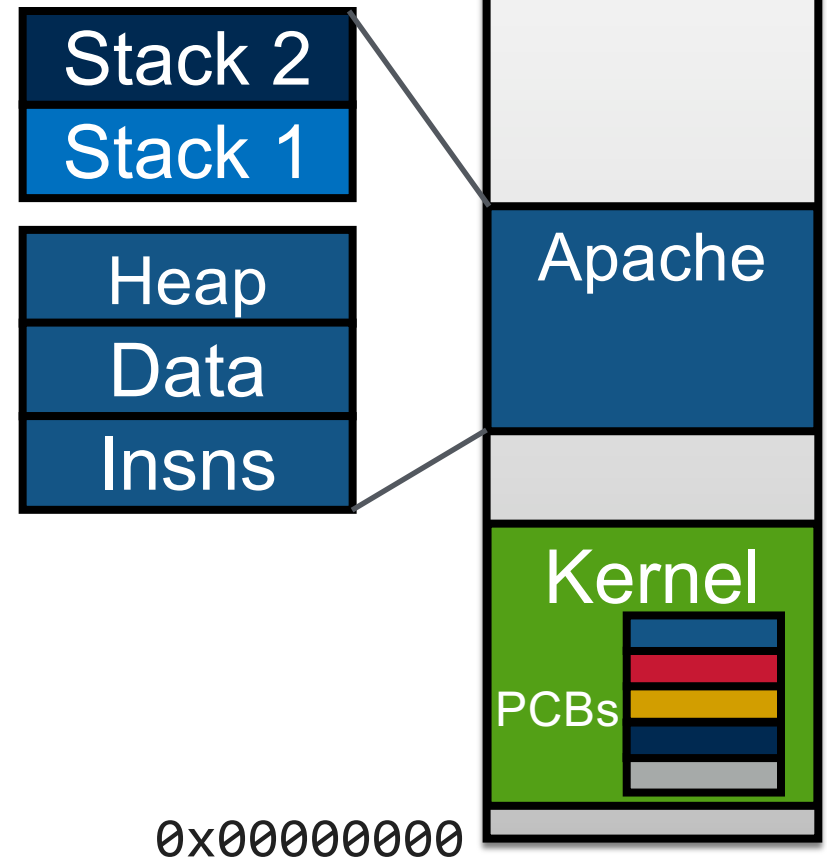
## One abstraction, two implementations:

1. “kernel threads”: each thread has its own PCB in the kernel, but the PCBs point to the same physical memory
2. “user threads”: one PCB for the process; threads implemented entirely in user space. Each thread has its own Thread Control Block (TCB) and context

# #1: Kernel-Level Threads

Kernel knows about, schedules threads (just like processes)

- Separate PCB for each thread
- PCBs have:
  - **same:** page table base register
  - **different:** PC, SP, registers, interrupt stack



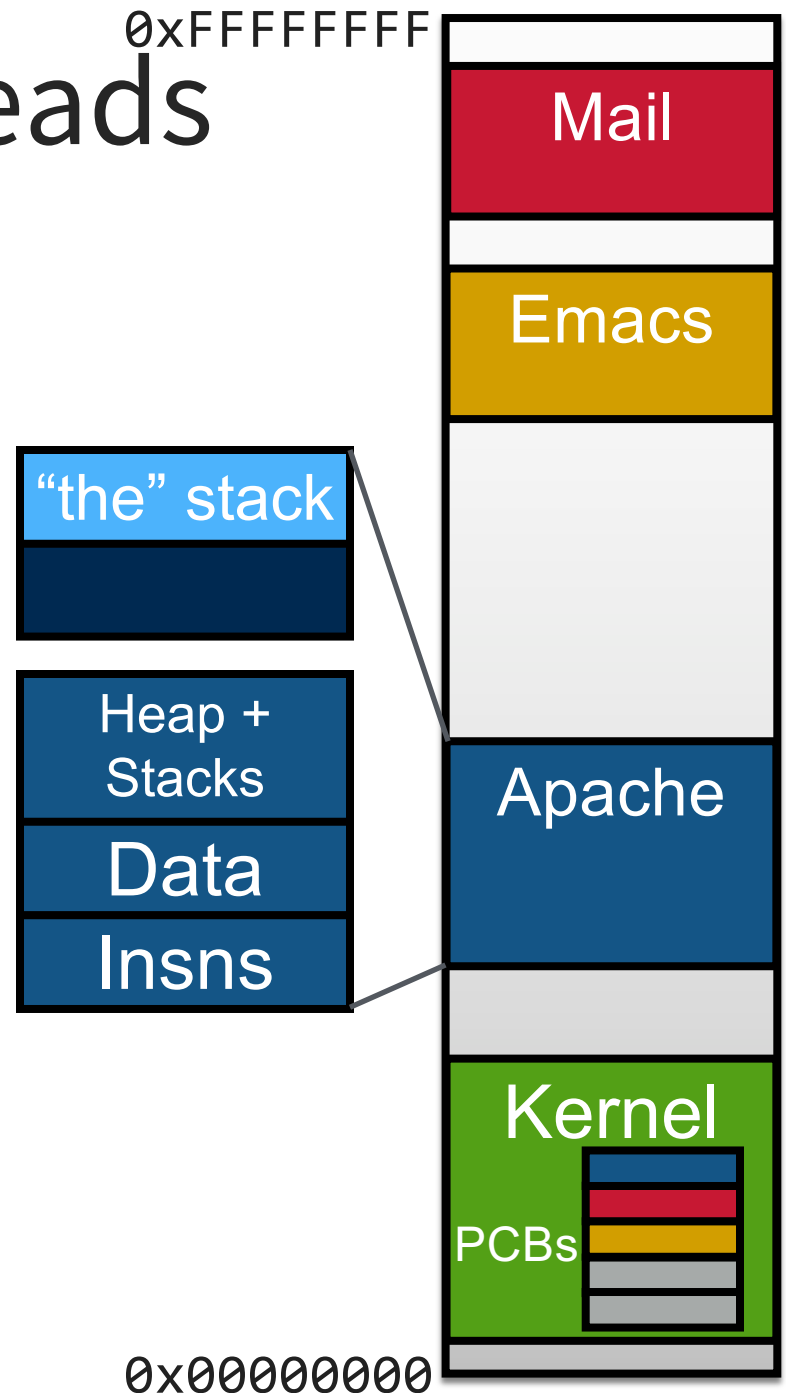
# #2: User-Level Threads

Run mini-OS in user space

- Real OS unaware of threads
- Single PCB
- Thread Control Block (TCB) for each thread

Generally more efficient than kernel-level threads (Why?)

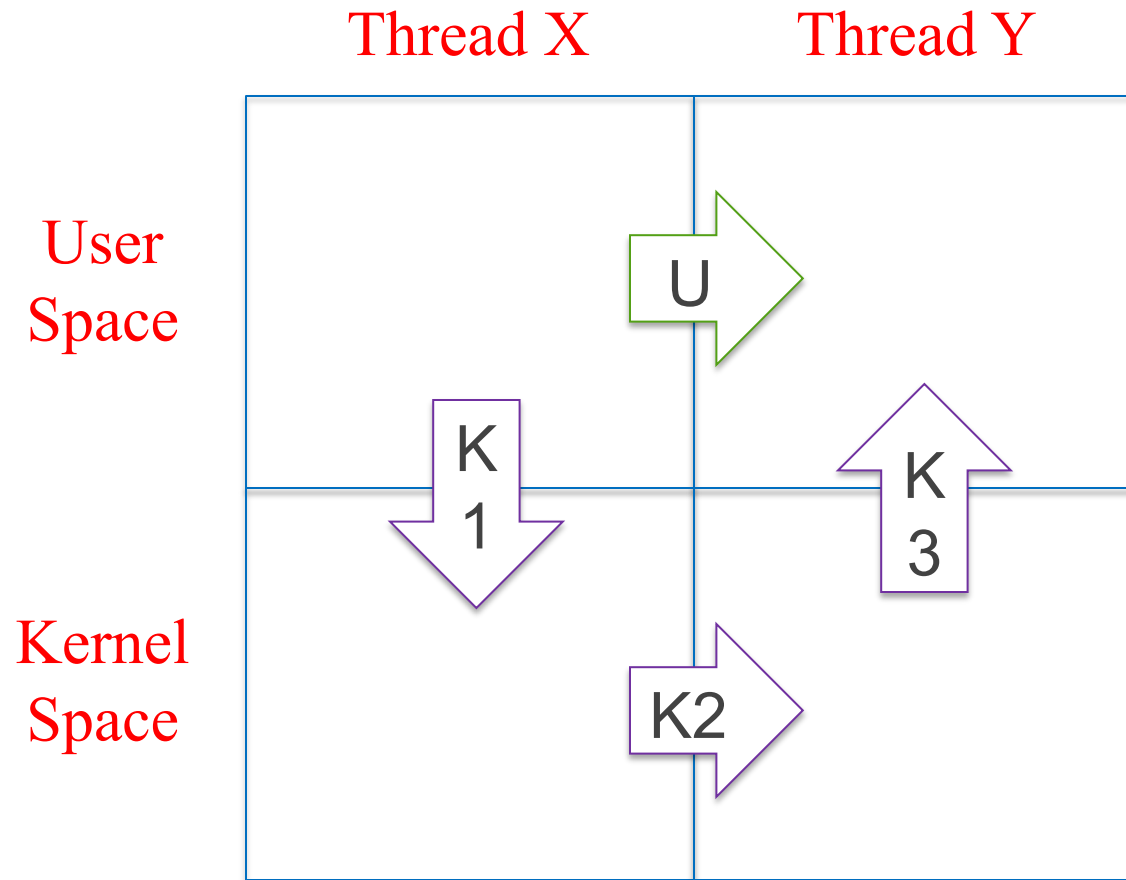
But kernel-level threads simplify system call handling (Why?)



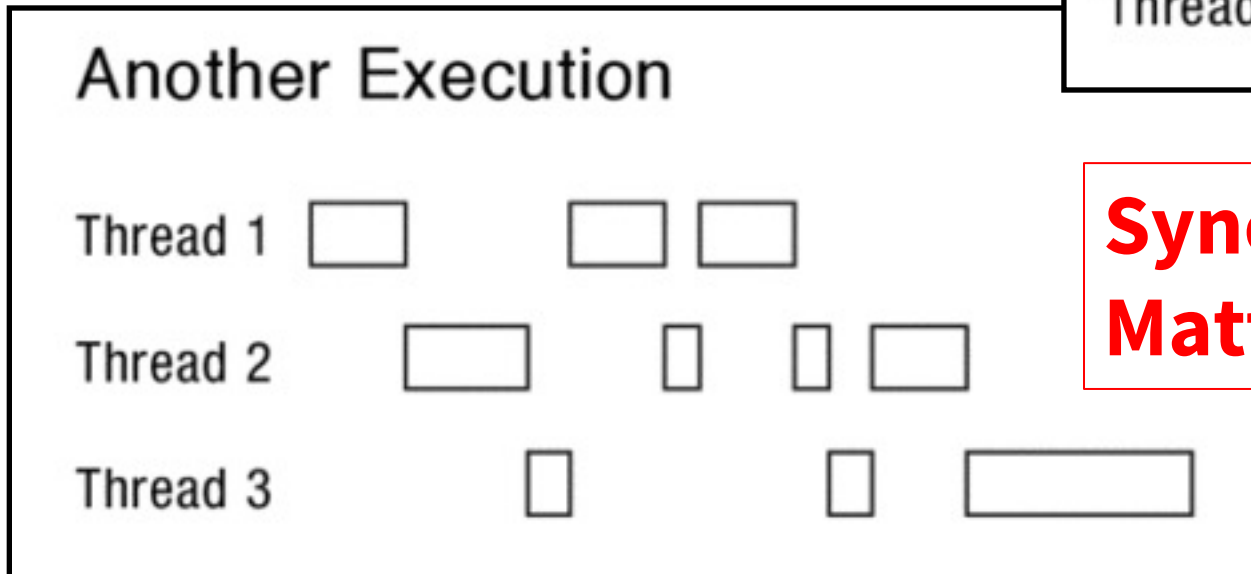
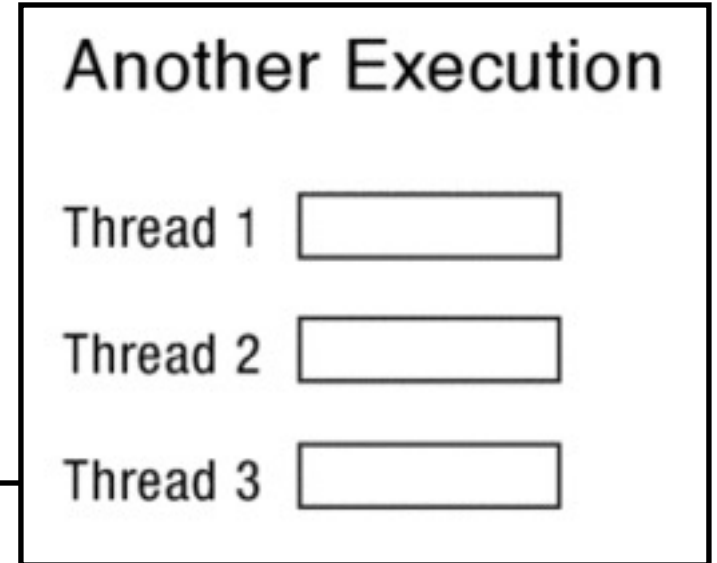
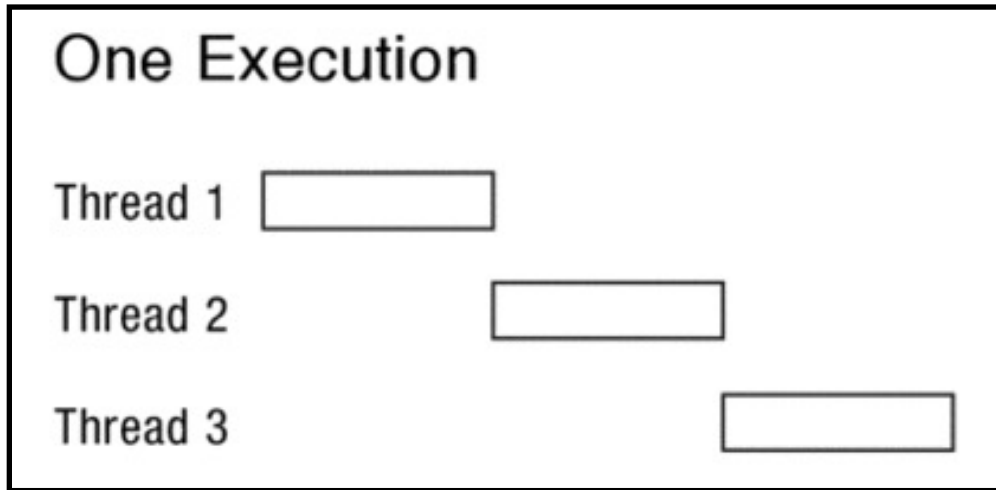
# Kernel- vs User-level Threads

Kernel-Level Threads	User-level Threads
<ul style="list-style-type: none"><li>• Easy to implement: just processes with shared page table</li></ul>	<ul style="list-style-type: none"><li>• Requires user-level context switches, scheduler</li></ul>
<ul style="list-style-type: none"><li>• Threads can run blocking system calls concurrently</li></ul>	<ul style="list-style-type: none"><li>• Blocking system call blocks all threads: needs O.S. support for non-blocking system calls</li></ul>
<ul style="list-style-type: none"><li>• Thread switch requires three context switches</li></ul>	<ul style="list-style-type: none"><li>• Thread switch efficiently implemented in user space</li></ul>

# Kernel vs User Thread Switch



# Do **not** presume to know the schedule



**Synchronization  
Matters!**

# Thread- vs Event-Based Programming

# What is “event-based” programming?

- No “blocking” operations
  - No `read()`, `wait()`, `lock()`, etc.
  - (in the extreme case)
- Code is a collection of event handlers
  - Invoked when some event happens
  - Runs to completion
    - Remember, no blocking operations
  - (Similar to I/O interrupt handlers)



# Example: event-based web server

**handler** *client\_request*(*client*, *URI*):

*contents* := *CACHE*[*URI*];

**if** *contents* != **None**:

*send*(*client*, *contents*);

**else**:

**if** *PENDING*[*URI*] == { }:

*start\_load\_file*(*URI*, *file\_loaded\_handler*);

*PENDING*[*URI*]  $\cup$  = { *client* };

**handler** *file\_loaded* (*URI*, *contents*):

*CACHE*[*URI*] := *contents*;

**for each** *client* **in** *PENDING*[*URI*]:

*send*(*client*, *contents*);

*PENDING*[*URI*] = { };

# Contrast: thread-based web server

**thread** client\_handler():

**for ever:**

```
(client, URI) = receive();           # blocks
CACHE.lock();                         # may block
while CACHE[URI] == None:
    NEEDED.lock(); NEEDED.U = { URI };
    NEEDED.notify(); NEEDED.unlock();
    CACHE.wait();                     # blocks
send(client, CACHE[URI]);
CACHE.unlock();
```

**thread** file\_loader(URI, contents):

**for ever:**

```
NEEDED.lock();                       # may block
while NEEDED == { }: NEEDED.wait();  # blocks
uris = NEEDED; NEEDED = { };
NEEDED.unlock();
for each URI in uris:
    contents = read(URI);             # blocks
    CACHE.lock(); CACHE[URI] = contents;
    CACHE.notifyAll(); CACHE.unlock();
```

# Decades old debate...

- Example debate papers
  - 1995: *Why Threads are a Bad Idea (for most purposes)*
    - J. Ousterhout (UC Berkeley, Sun Labs, now at Stanford)
  - 2003: *Why Events are a Bad Idea (for high-concurrency servers)*
    - R. van Behren, J. Condit, E. Brewer (UC Berkeley)
- But also known to be logically equivalent:
  - 1978, *On the Duality of Operating Systems Structures*
    - H.C. Lauer, R.M. Needham

# Pros and Cons

Event-Based	Thread-Based
good for I/O-parallelism/GUIs	good for any parallelism
no context switch overhead (contexts are short-lived)	keeps track of control flow
does not need locks	needs locks
code becomes spaghetti	code relatively easy to read
deterministic, easy to debug	hard to debug ( <i>Harmony</i> to the rescue :-)

# Shell

# What is a Shell?

- is an interpreter (i.e., just another program)
- language allows user to create/manage programs
- sh            Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
- bash        “Bourne-Again” Shell

*Runs at user-level. Uses syscalls: fork, exec, etc.*

# What is a Shell?

- Reads lines of input
  - command [arg1 ...]
- And executes them
- Full programming language in its own right
- e.g.:

```
$ for x in a b c
```

```
> do echo $x
```

```
> done
```

# echo is a print command

# Shell has state

- Just like other programming languages
- Includes:
  - home directory
  - working directory
  - list of processes that are running
- Commands often modify the state



# Some important commands

- `echo [args]`           # print arguments
- `ls`                    # list the working directory
- `pwd`                  # print working directory
- `cd [dir]`             # change working directory
  - default is “home” directory
- `ps`                    # list running processes

# “flags” (aka options)

- arguments to command that start with ‘-’
- examples:
  - `ls -l`           # long listing
  - `ps -a`           # print all processes

# “foreground” vs. “background”

The shell either

- is reading from standard input
- is waiting for a process to finish
  - this is the *foreground* process
  - other processes are *background processes*
- To start a background process, add ‘&’
- e.g.:
  - (sleep 5; echo hello)&
  - x & y                    # runs x in background and y in foreground

Background processes should not read from standard input!

Why not?

# Pipelines

- `x | y`
  - runs both `x` and `y` in foreground
  - output of `x` is input to `y`
  - finishes when both `x` and `y` finish
- e.g.:
  - `echo robbert | tr b B`