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]

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:

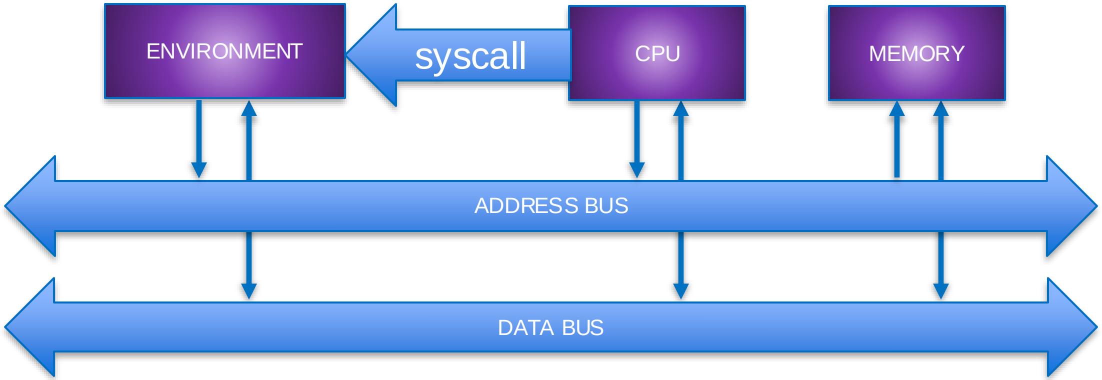
- executable code
  - CPU instructions
- data
  - information manipulated by these instructions

- 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



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 ≠ Program

A program is passive:

code + data

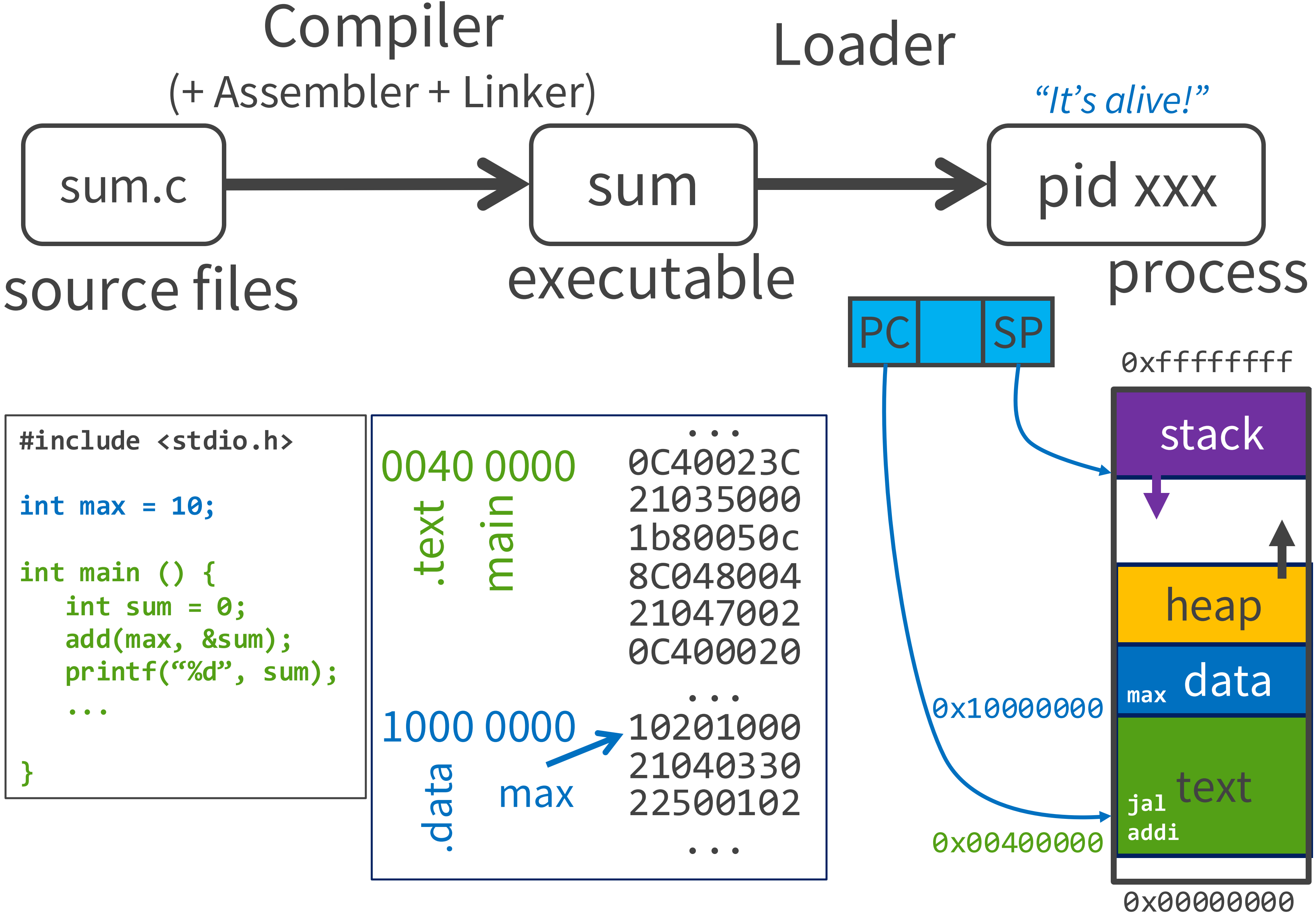
A process is ***alive:***

changes data + registers + files + …

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

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

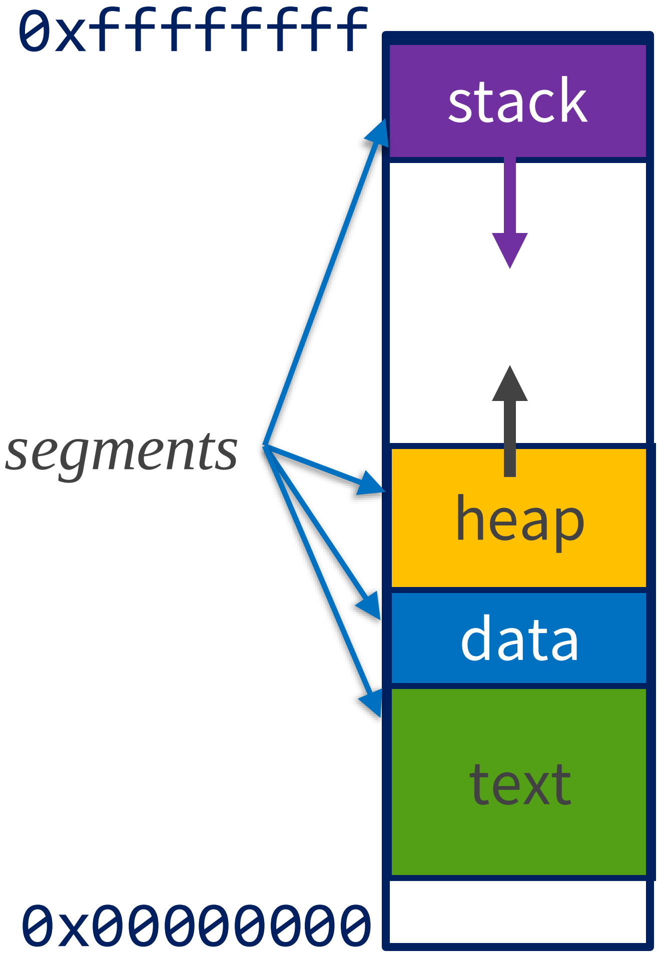
A Day in the Life of a Program



```
#include <stdio.h>

int max = 10;

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

}
```

```
                  ...
0040 0000      0C40023C
               21035000
               1b80050c
               8C048004
               21047002
               0C400020
                  ...
1000 0000      10201000
               21040330
               22500102
                  ...
```

Logical view of process memory



call stack

heap used for memory allocation (malloc)

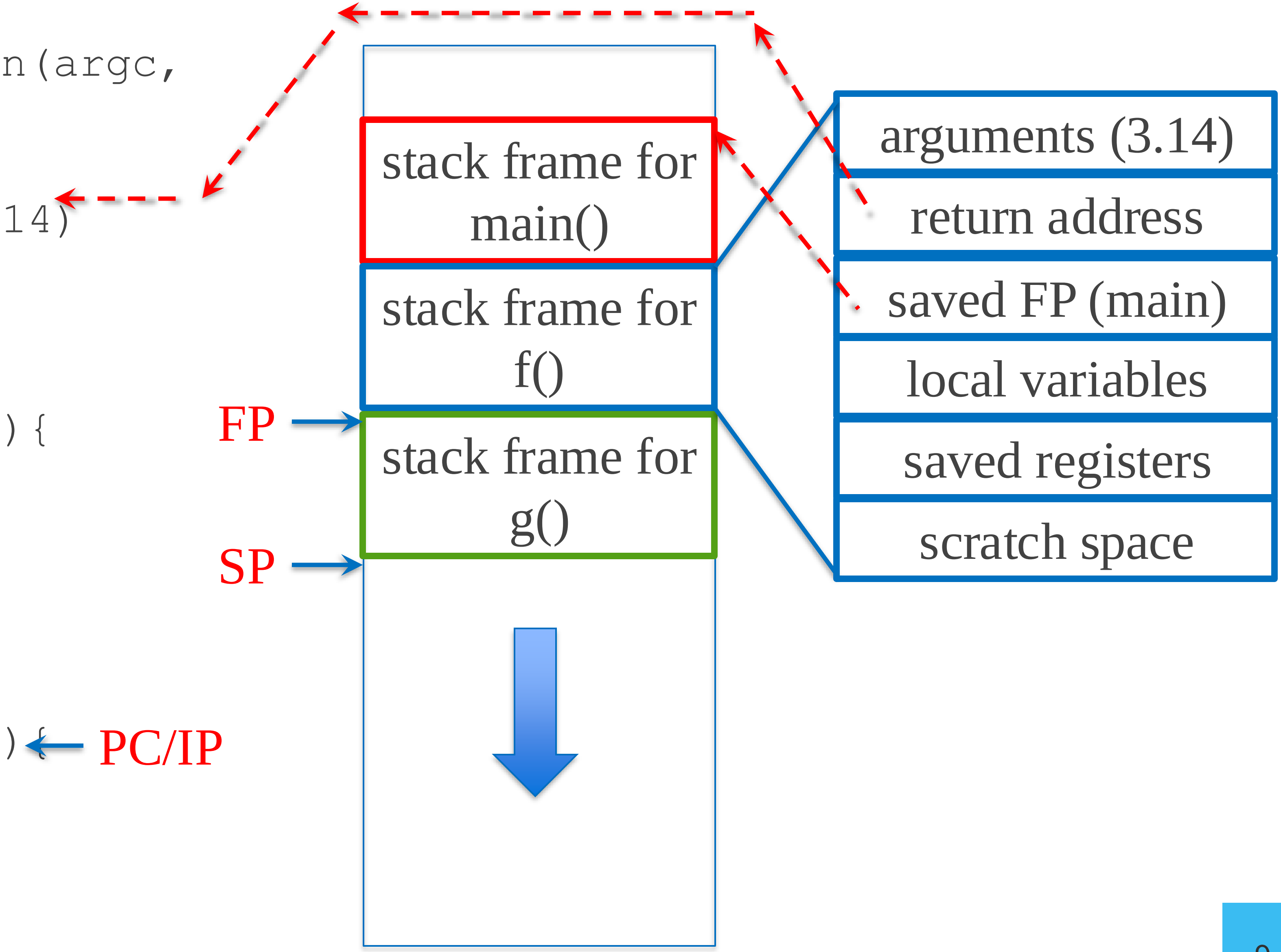
data segment contains global variables

read-only text segment contains code and constants

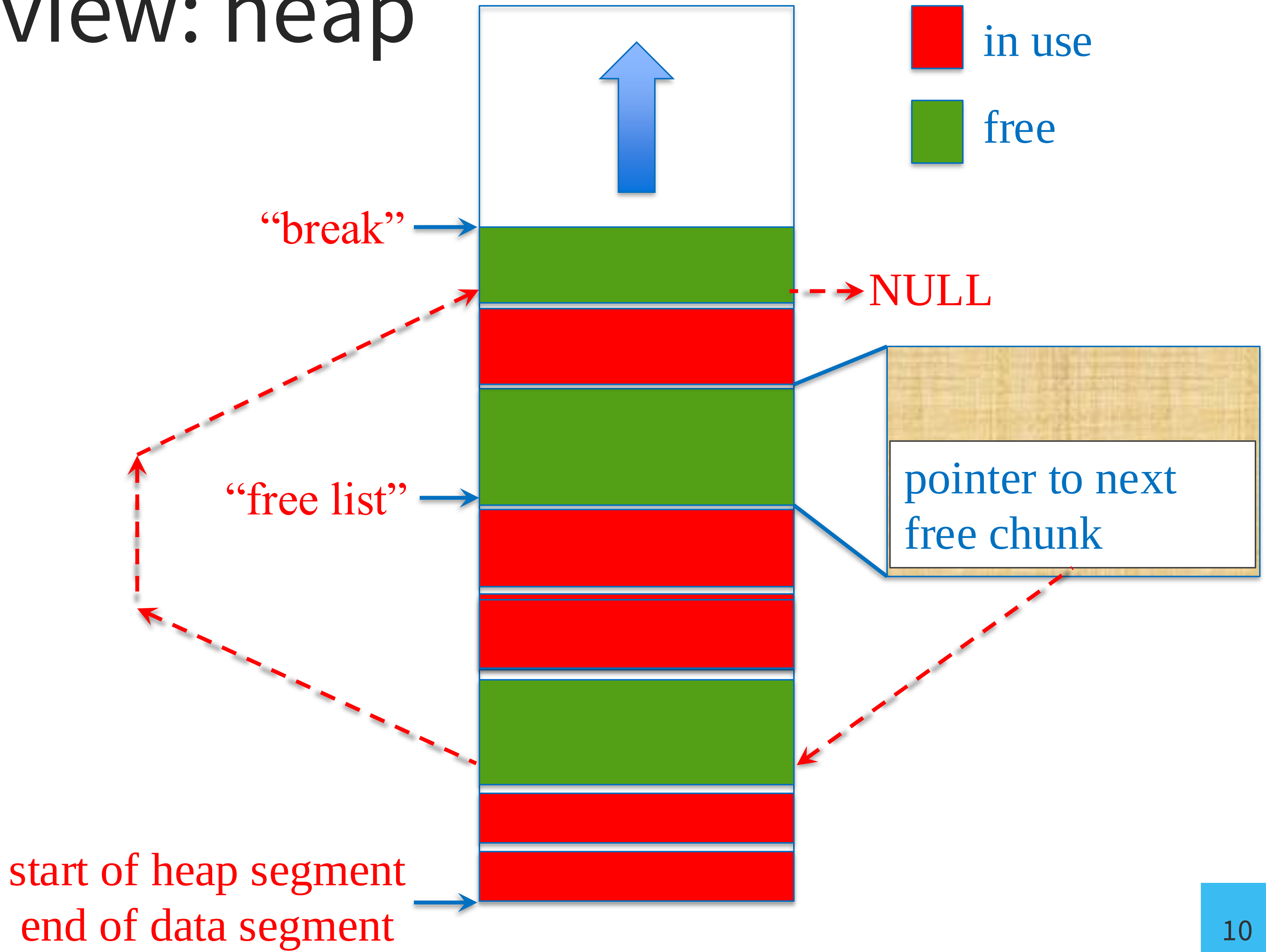
How many bits in an address for this CPU?
Why is address 0 not mapped?

Review: stack (aka call stack)

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

int f(x){
    …
    g();
    …
}

int g(y){
    …
}
```



Review: heap



🟥 in use

🟩 free

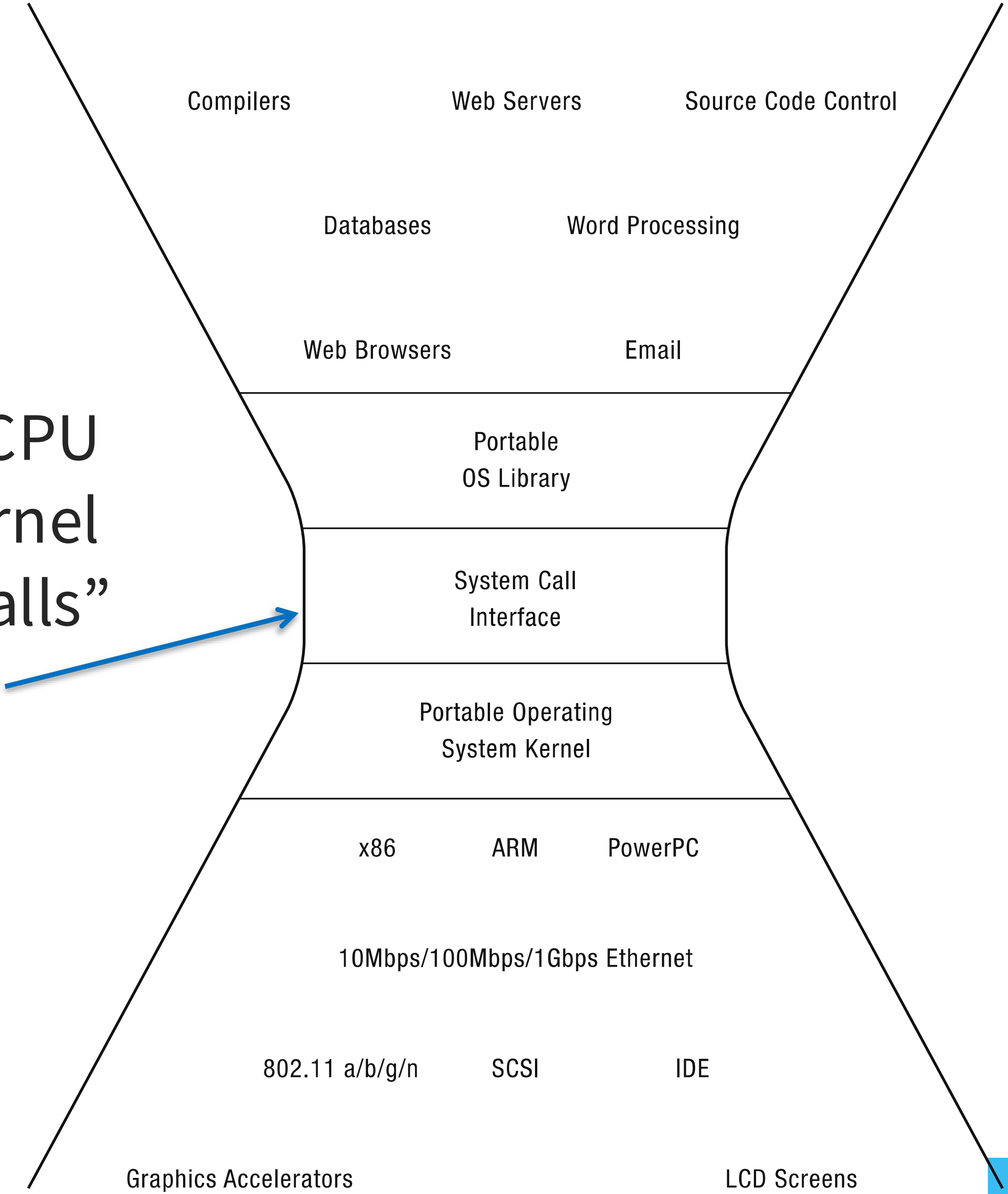
Environment

- 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

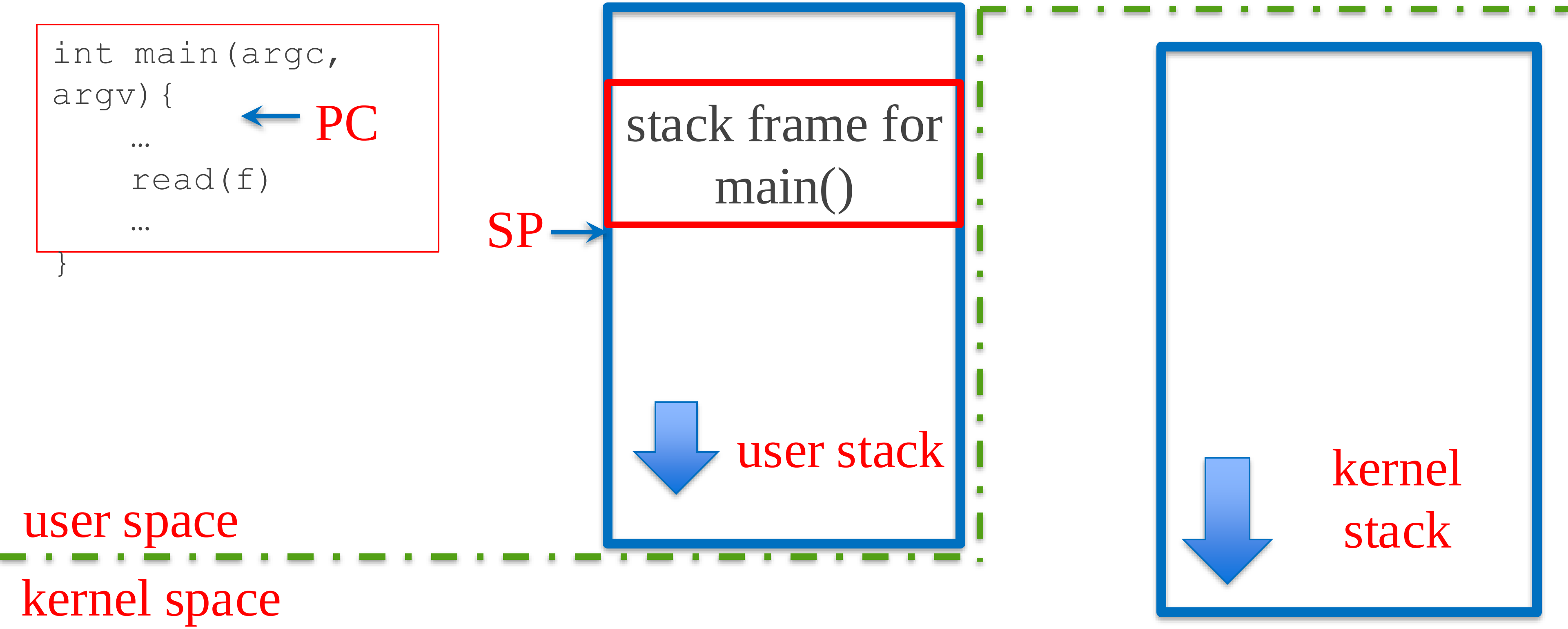
Kernel:

5. Executes `syscall` interrupt handler
6. Places result in dedicated register
7. Executes `return_from_interrupt`

Process:

8. Executes `return_from_function`

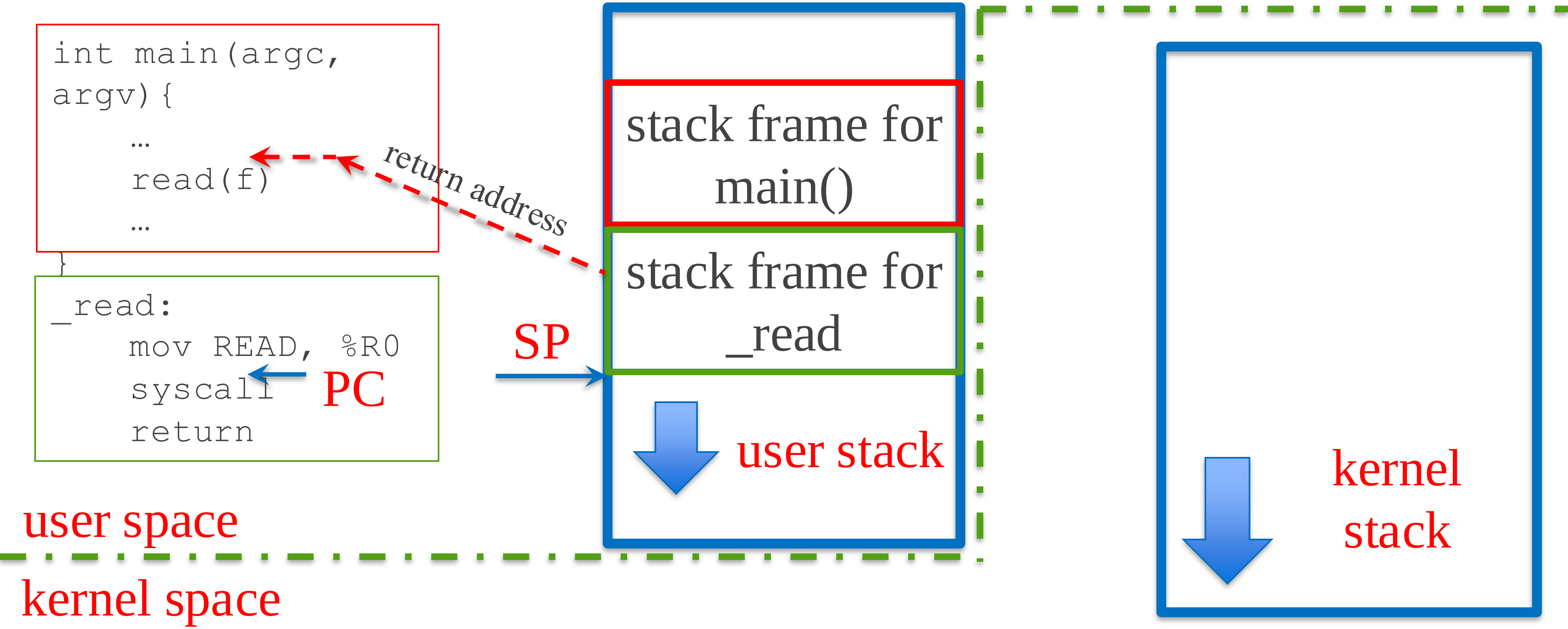
Executing `read` System Call



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

note kernel stack empty while process running

Executing `read` System Call

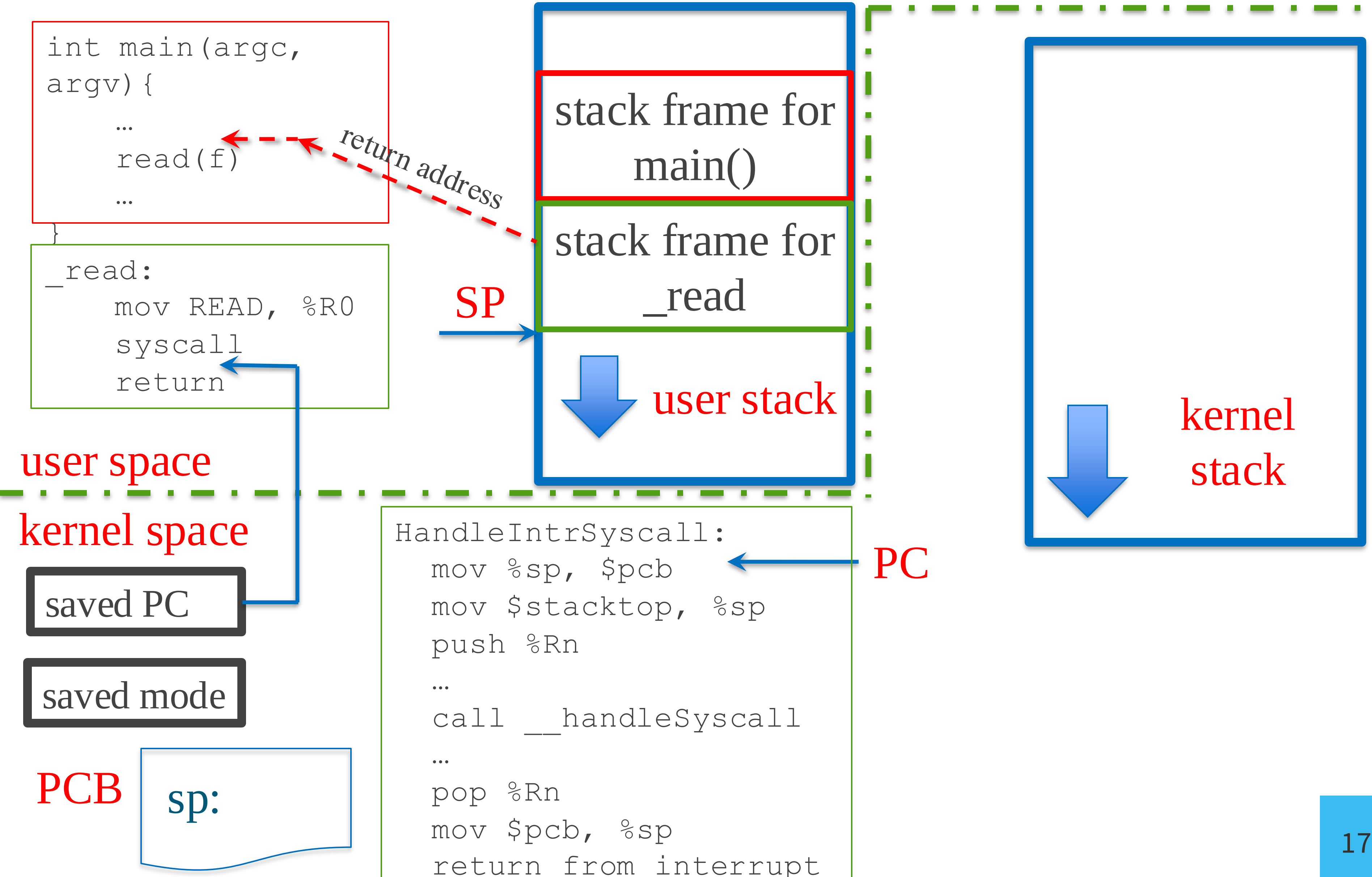


```
int main(argc,
argv){
     …
     read(f)
     …
}
```

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

note kernel stack empty while process running

Executing `read` System Call

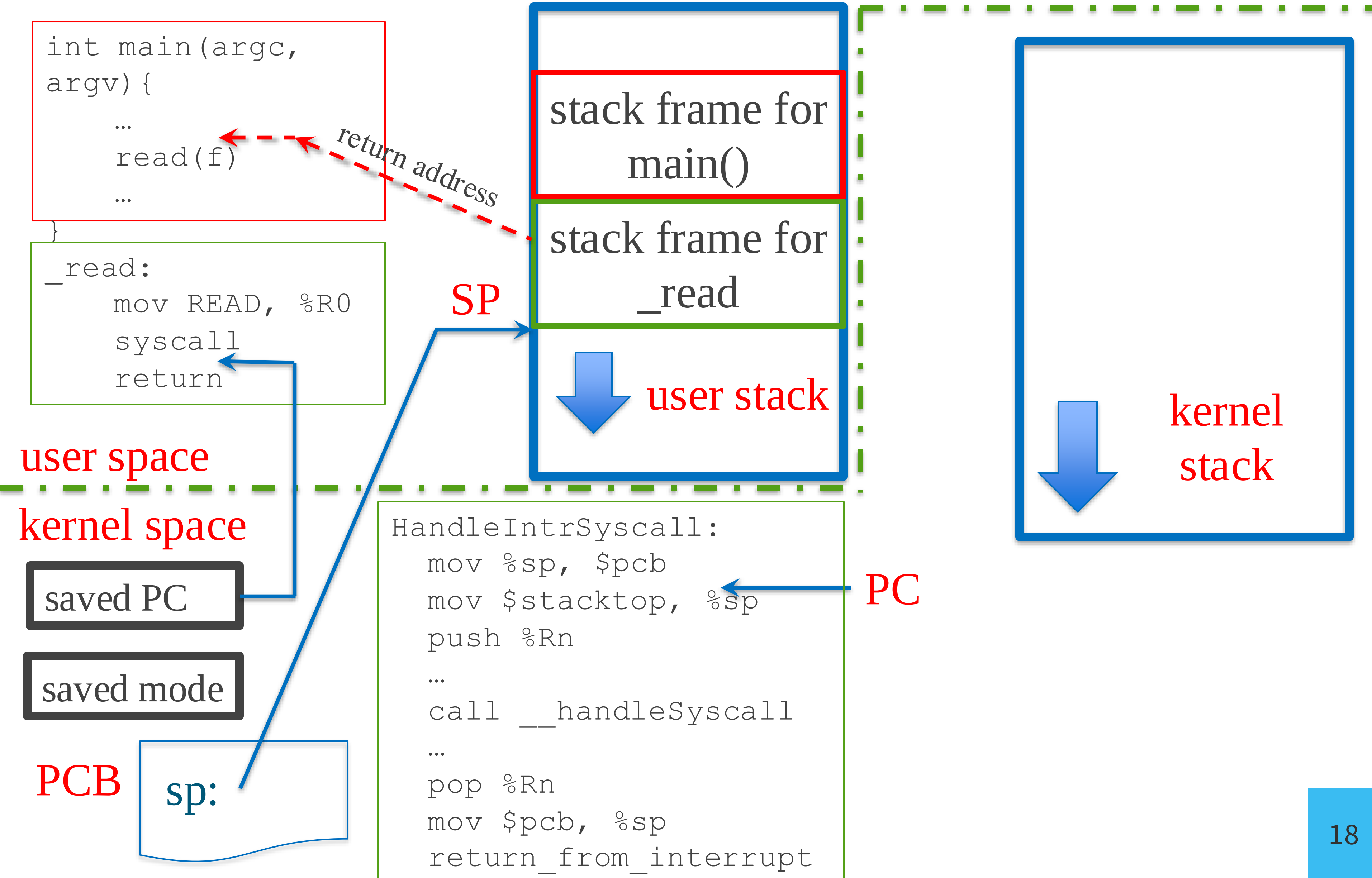


```
int main(argc,
argv){
    …
    read(f)
    …
}
```

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

```
HandleIntrSyscall:
    mov %sp, $pcb
    mov $stacktop, %sp
    push %Rn
    …
    call __handleSyscall
    …
    pop %Rn
    mov $pcb, %sp
    return_from_interrupt
```

Executing `read` System Call

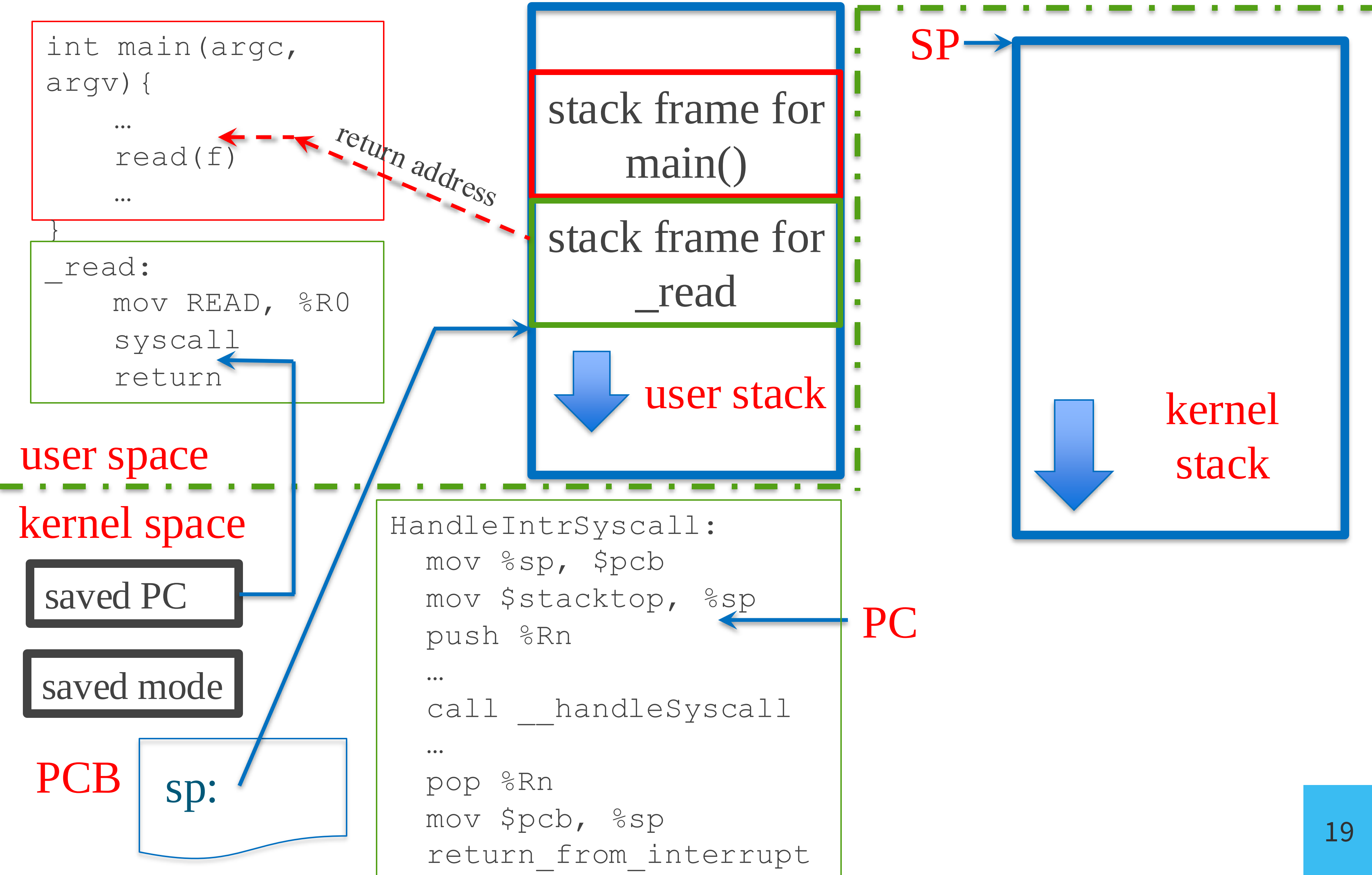


```
int main(argc, argv){
    …
    read(f)
    …
}
```

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

```
HandleIntrSyscall:
  mov %sp, $pcb
  mov $stacktop, %sp
  push %Rn
  …
  call __handleSyscall
  …
  pop %Rn
  mov $pcb, %sp
  return_from_interrupt
```

Executing `read` System Call

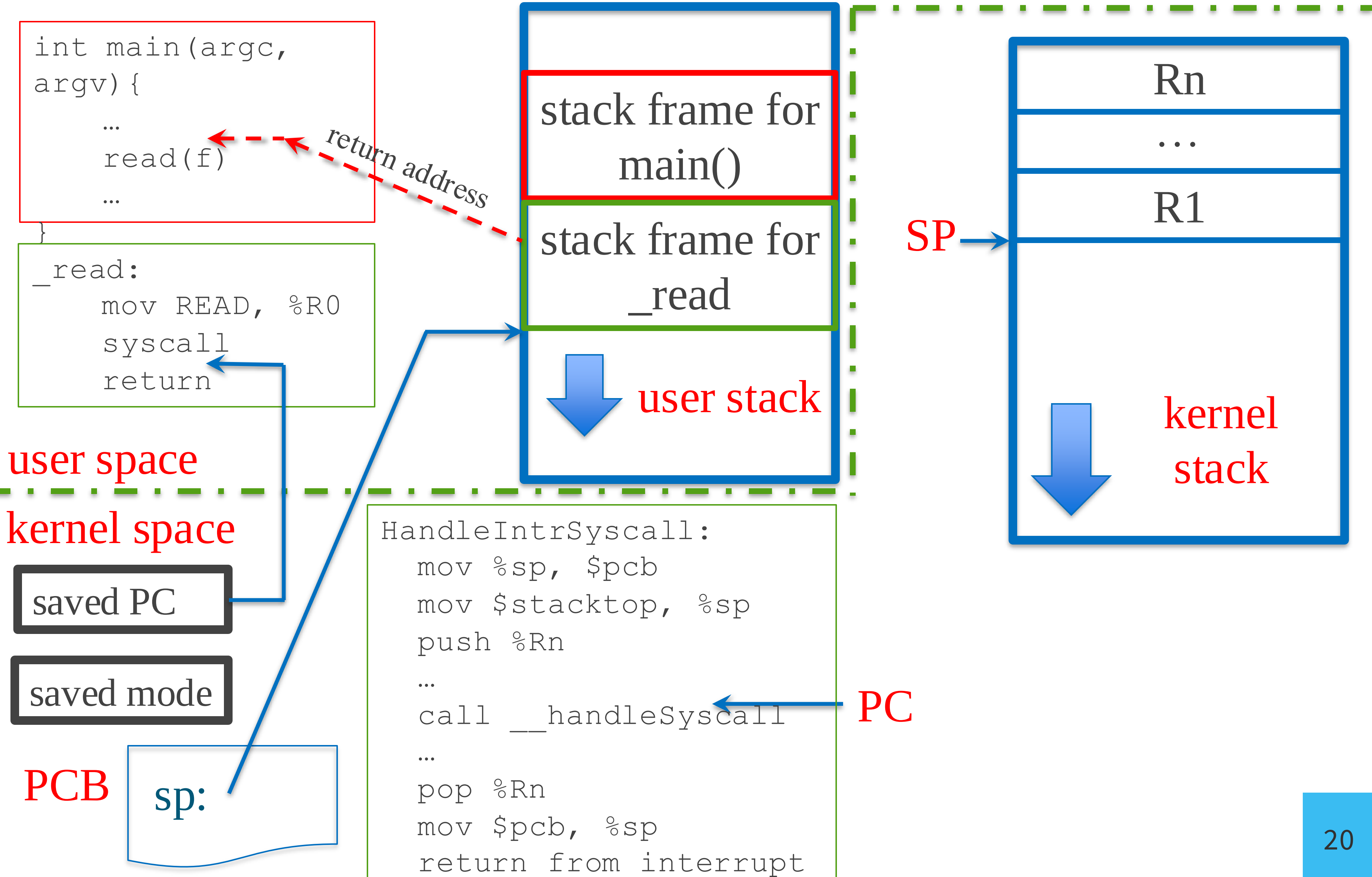


```
int main(argc,
argv){
    …
    read(f)
    …
}
```

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

```
HandleIntrSyscall:
   mov %sp, $pcb
   mov $stacktop, %sp
   push %Rn
   …
   call __handleSyscall
   …
   pop %Rn
   mov $pcb, %sp
   return_from_interrupt
```

Executing `read` System Call

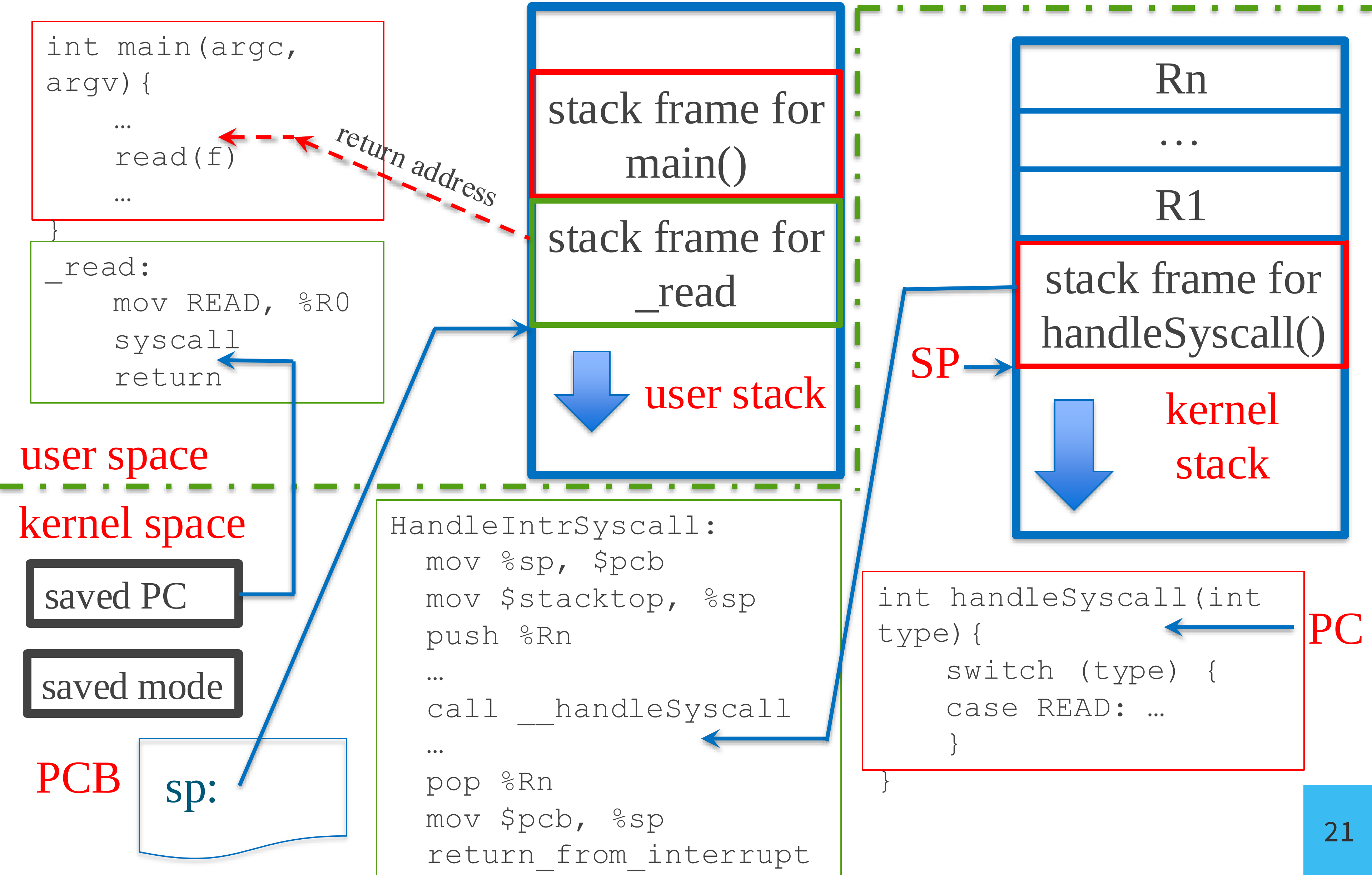


```
int main(argc,
argv){
    …
    read(f)
    …
}
```

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

```
HandleIntrSyscall:
  mov %sp, $pcb
  mov $stacktop, %sp
  push %Rn
  …
  call __handleSyscall
  …
  pop %Rn
  mov $pcb, %sp
  return_from_interrupt
```

Executing `read` System Call



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

return address

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

stack frame for main()

stack frame for _read

user stack

user space

kernel space

saved PC

saved mode

PCB

sp:

Rn

…

R1

stack frame for handleSyscall()

SP

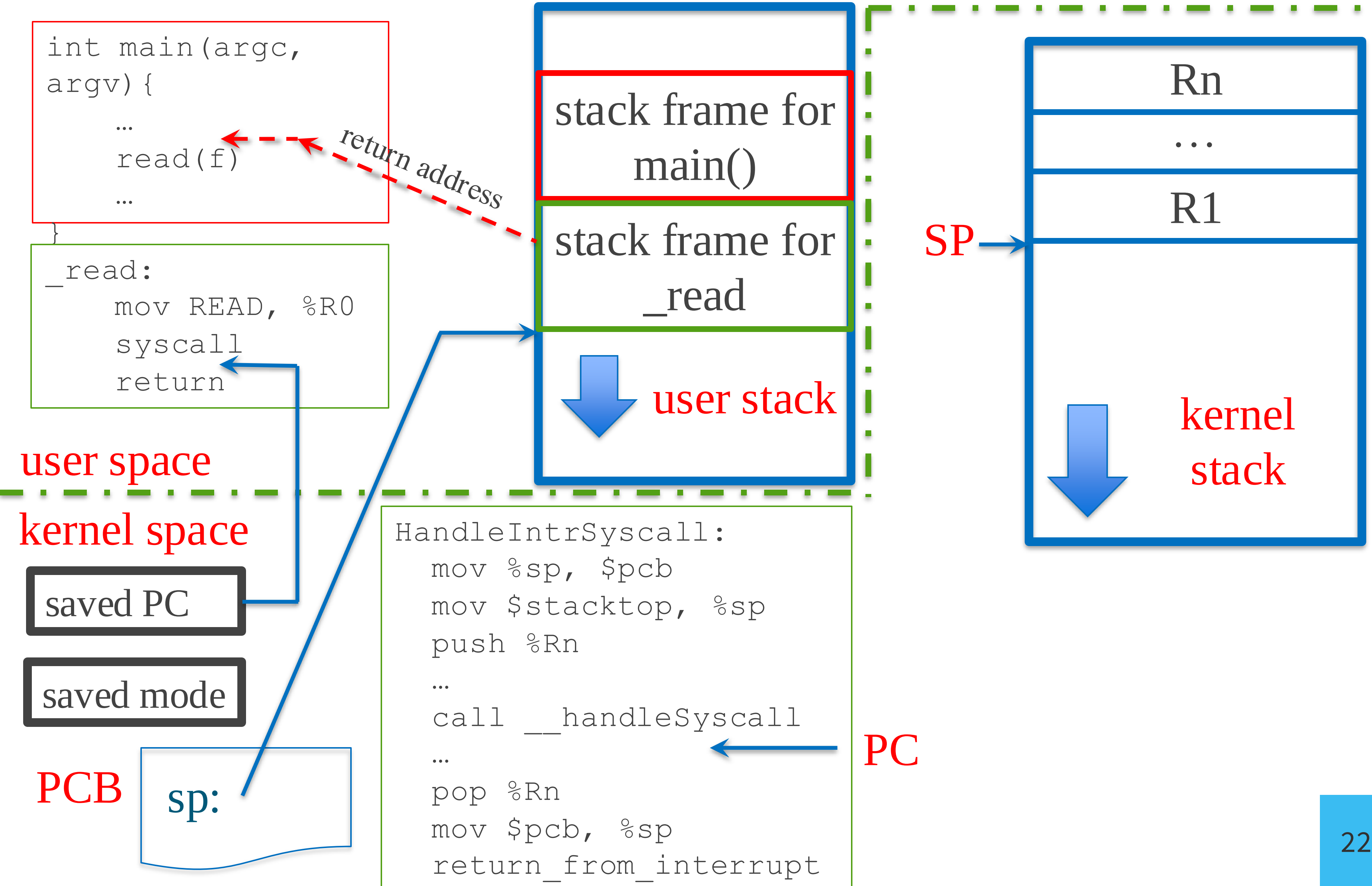
kernel stack

```
HandleIntrSyscall:
  mov %sp, $pcb
  mov $stacktop, %sp
  push %Rn
  …
  call __handleSyscall
  …
  pop %Rn
  mov $pcb, %sp
  return_from_interrupt
```

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

PC

Executing `read` System Call



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

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

return address

stack frame for main()

stack frame for _read

user stack

user space

kernel space

saved PC

saved mode

PCB

sp:

```
HandleIntrSyscall:
   mov %sp, $pcb
   mov $stacktop, %sp
   push %Rn
   …
   call __handleSyscall
   …
   pop %Rn
   mov $pcb, %sp
   return_from_interrupt
```

PC

SP

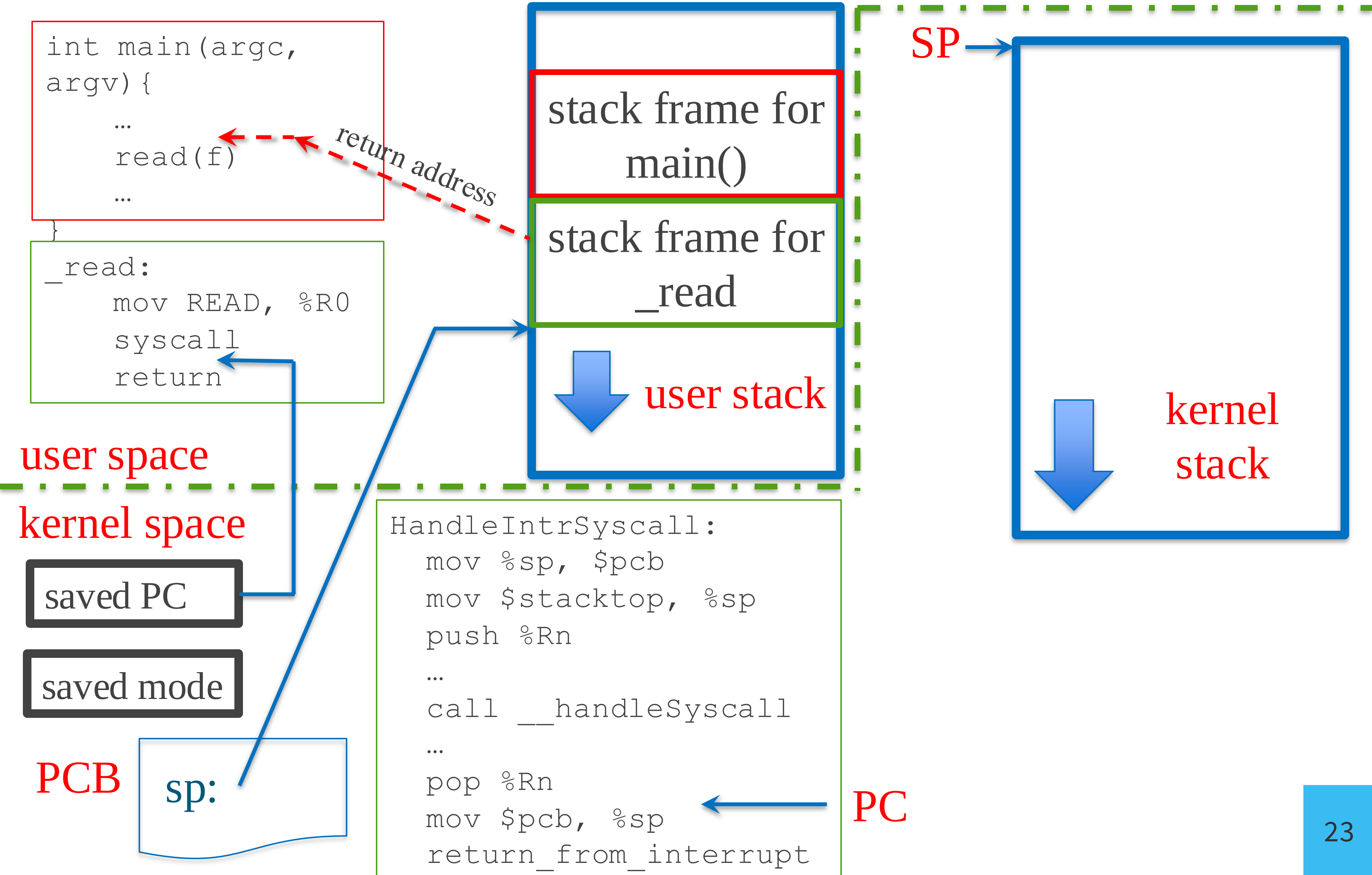
Rn

…

R1

kernel stack

Executing `read` System Call



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

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

return address

stack frame for main()

stack frame for _read

user stack

SP

kernel stack

user space

kernel space

saved PC

saved mode

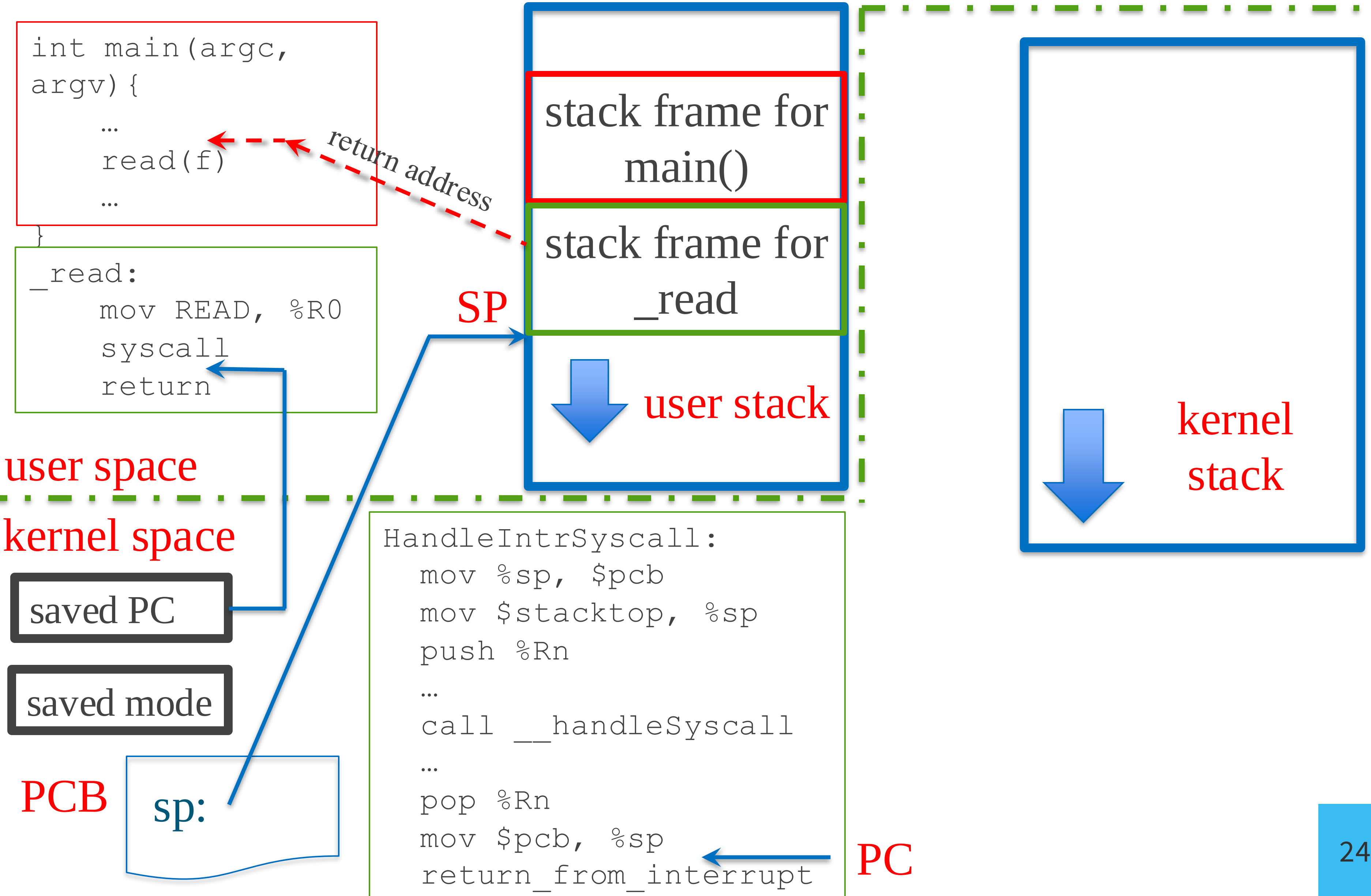
PCB

sp:

```
HandleIntrSyscall:
  mov %sp, $pcb
  mov $stacktop, %sp
  push %Rn
  …
  call __handleSyscall
  …
  pop %Rn
  mov $pcb, %sp
  return_from_interrupt
```

PC

Executing `read` System Call



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

return address

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

stack frame for main()

stack frame for _read

SP

user stack

kernel stack

user space

kernel space

saved PC

saved mode

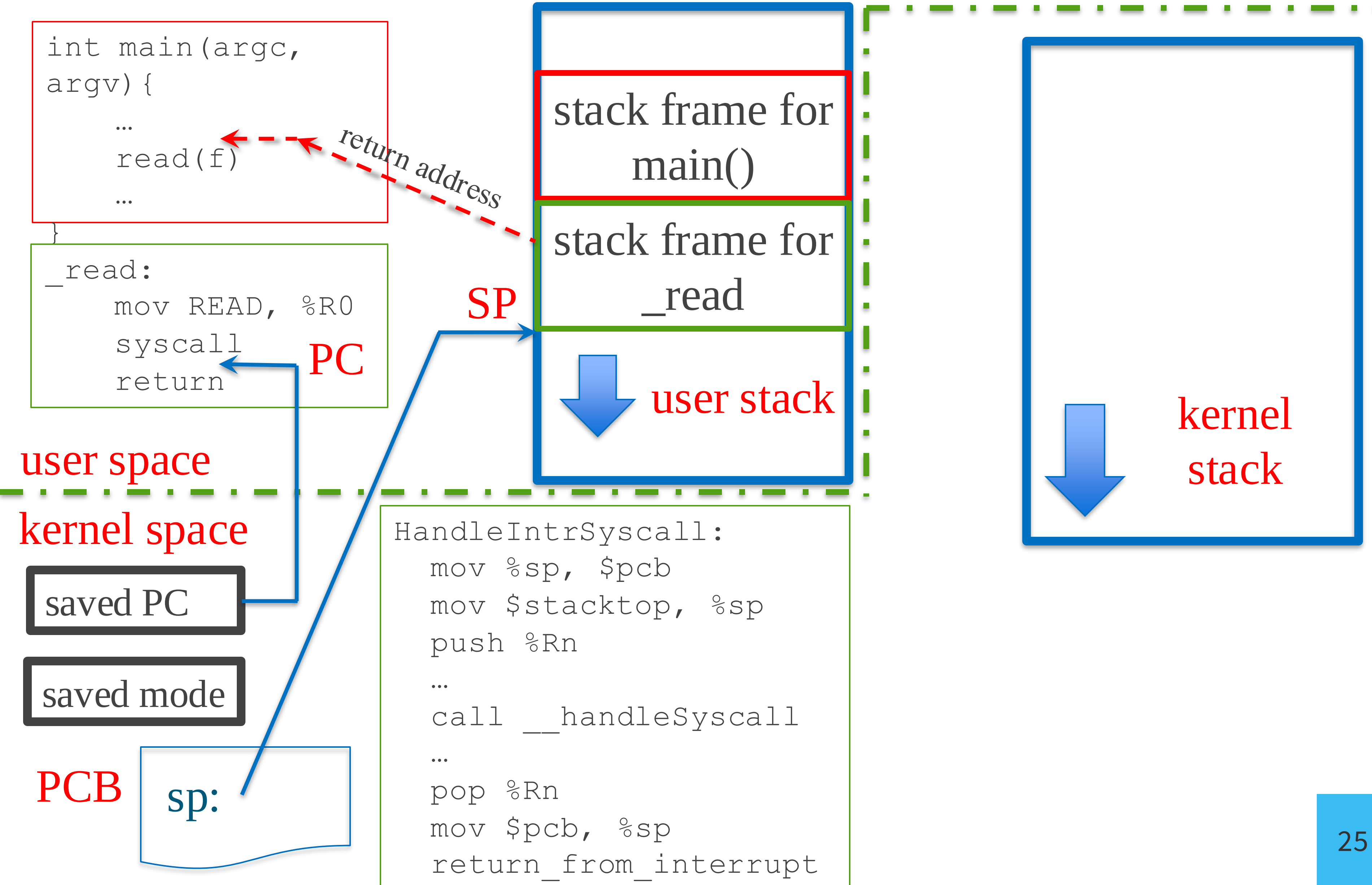
PCB

sp:

```
HandleIntrSyscall:
  mov %sp, $pcb
  mov $stacktop, %sp
  push %Rn
  …
  call __handleSyscall
  …
  pop %Rn
  mov $pcb, %sp
  return_from_interrupt
```

PC

Executing `read` System Call



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

return address

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

PC

SP

stack frame for main()

stack frame for _read

user stack

kernel stack

user space

kernel space

saved PC

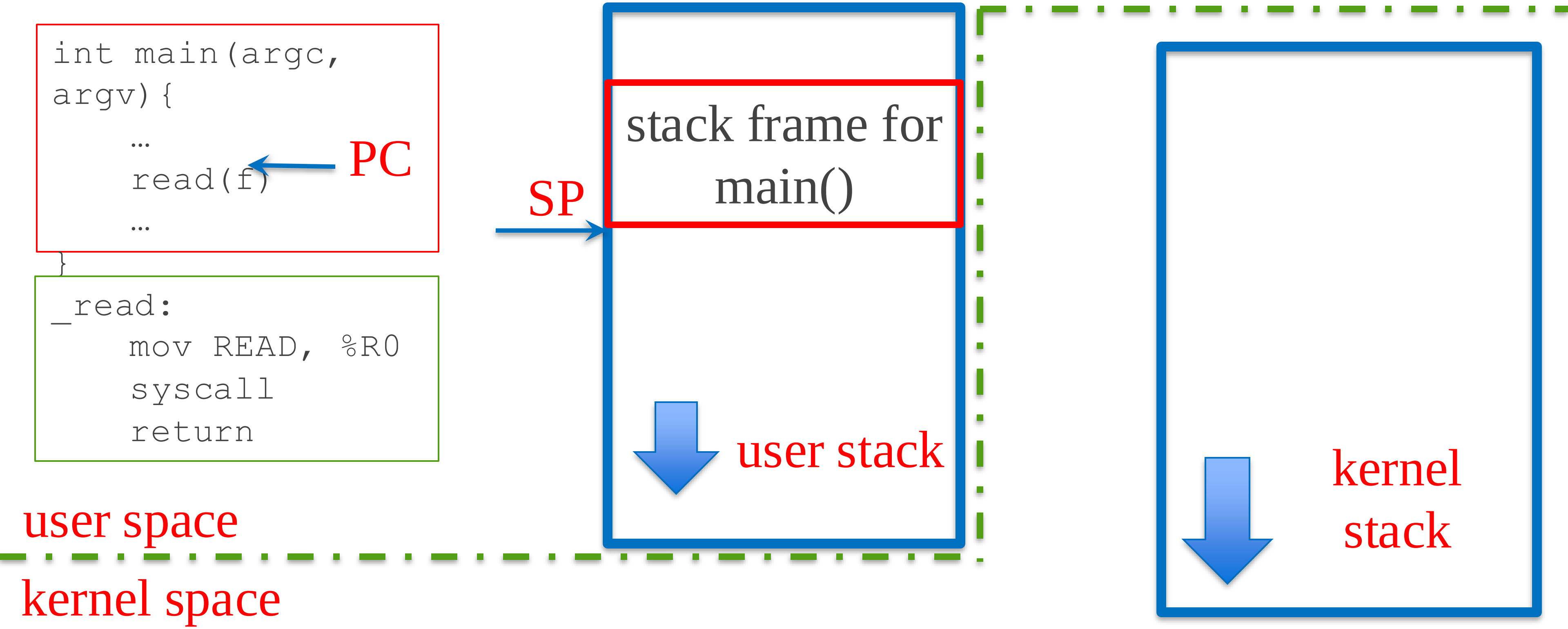
saved mode

PCB

sp:

```
HandleIntrSyscall:
  mov %sp, $pcb
  mov $stacktop, %sp
  push %Rn
  …
  call __handleSyscall
  …
  pop %Rn
  mov $pcb, %sp
  return_from_interrupt
```

Executing `read` System Call



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

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

PC

SP

stack frame for main()

user stack

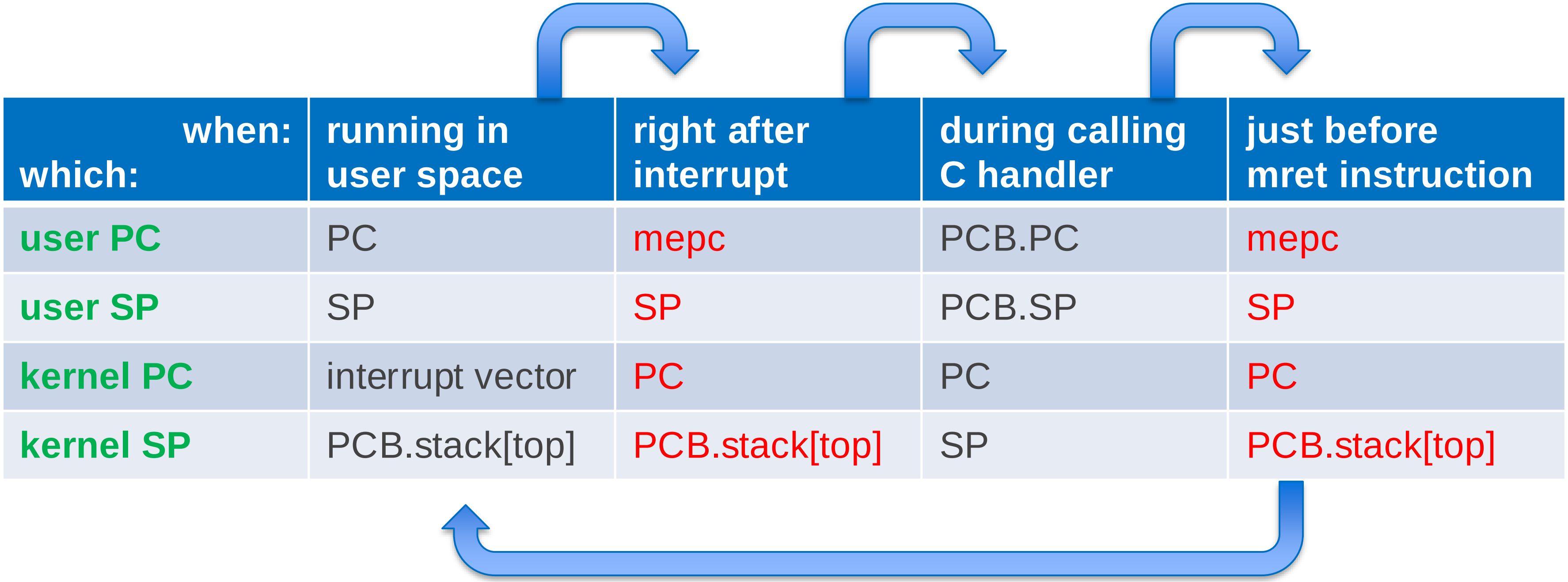
kernel stack

user space

kernel space

Keep your eye on the balls!

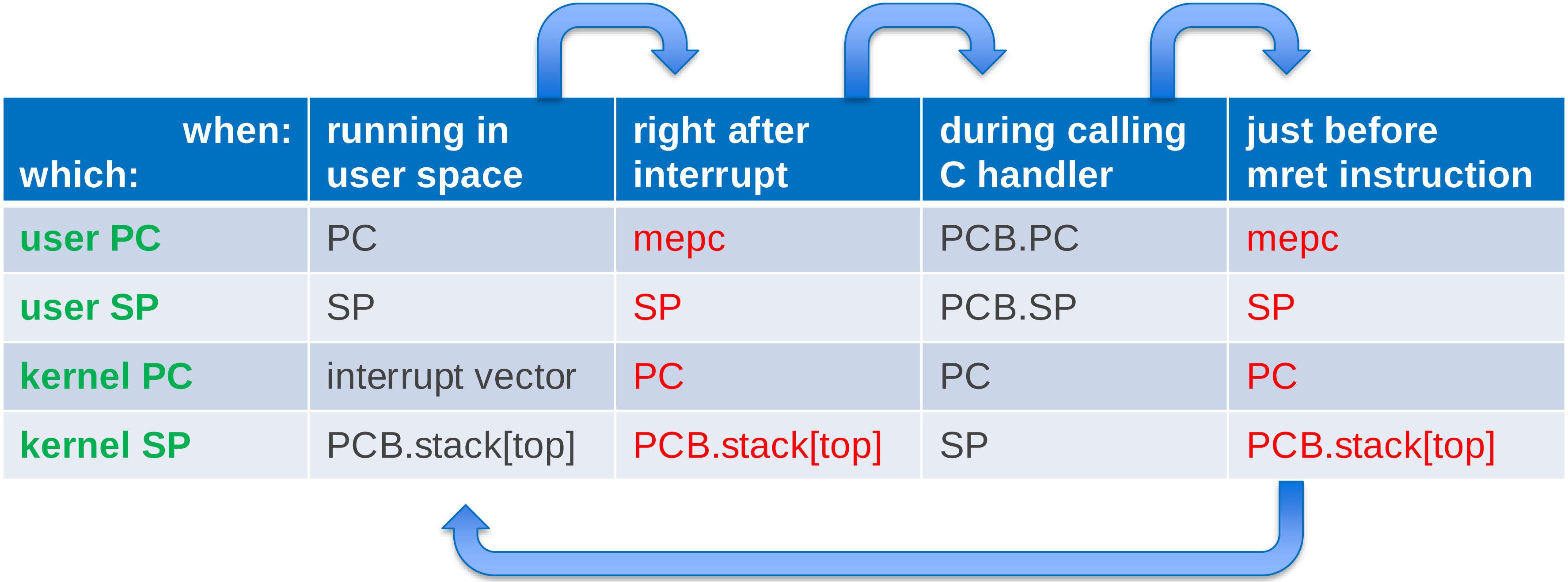
Where are the values of the virtual PC and SP registers (RISC-V)?



when: which:	running in user space	right after interrupt	during calling C handler	just before mret instruction
user PC	PC	mepc	PCB.PC	mepc
user SP	SP	SP	PCB.SP	SP
kernel PC	interrupt vector	PC	PC	PC
kernel SP	PCB.stack[top]	PCB.stack[top]	SP	PCB.stack[top]

Keep your eye on the balls!

Where are the values of the virtual PC and SP registers (RISC-V)?



when: which:	running in user space	right after interrupt	during calling C handler	just before mret instruction
user PC	PC	mepc	PCB.PC	mepc
user SP	SP	SP	PCB.SP	SP
kernel PC	interrupt vector	PC	PC	PC
kernel SP	PCB.stack[top]	PCB.stack[top]	SP	PCB.stack[top]

How about the general-purpose registers?

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!

(note: kernel should not block!!!)

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*

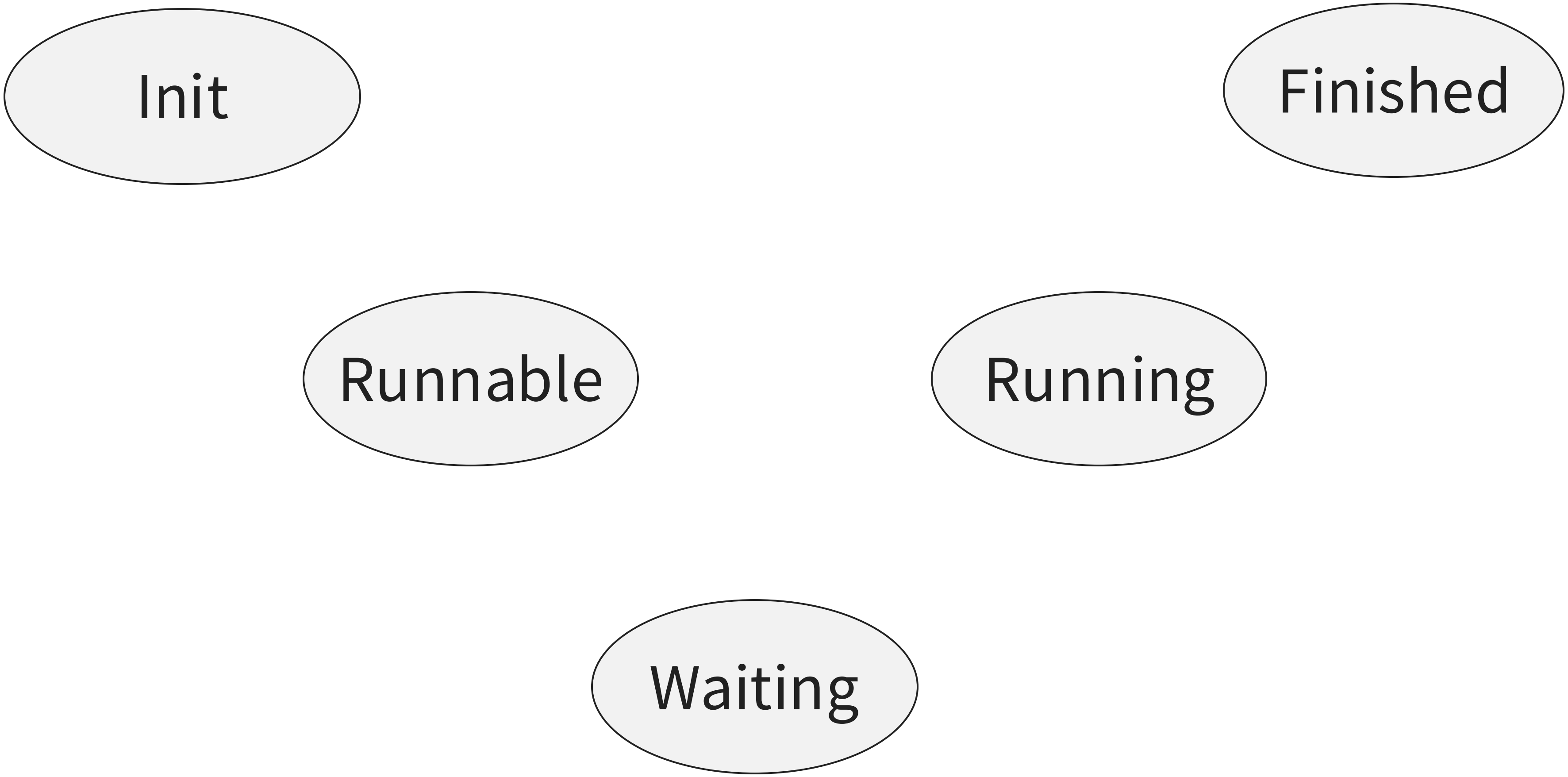
For now, assume we have a single core CPU

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
- kernel stack
- saved user SP
  - points into user stack
- saved kernel SP
  - points into kernel stack
  - kernel stack contains saved registers and kernel call stack for this process
- *… and more!*

Process Life Cycle



Init

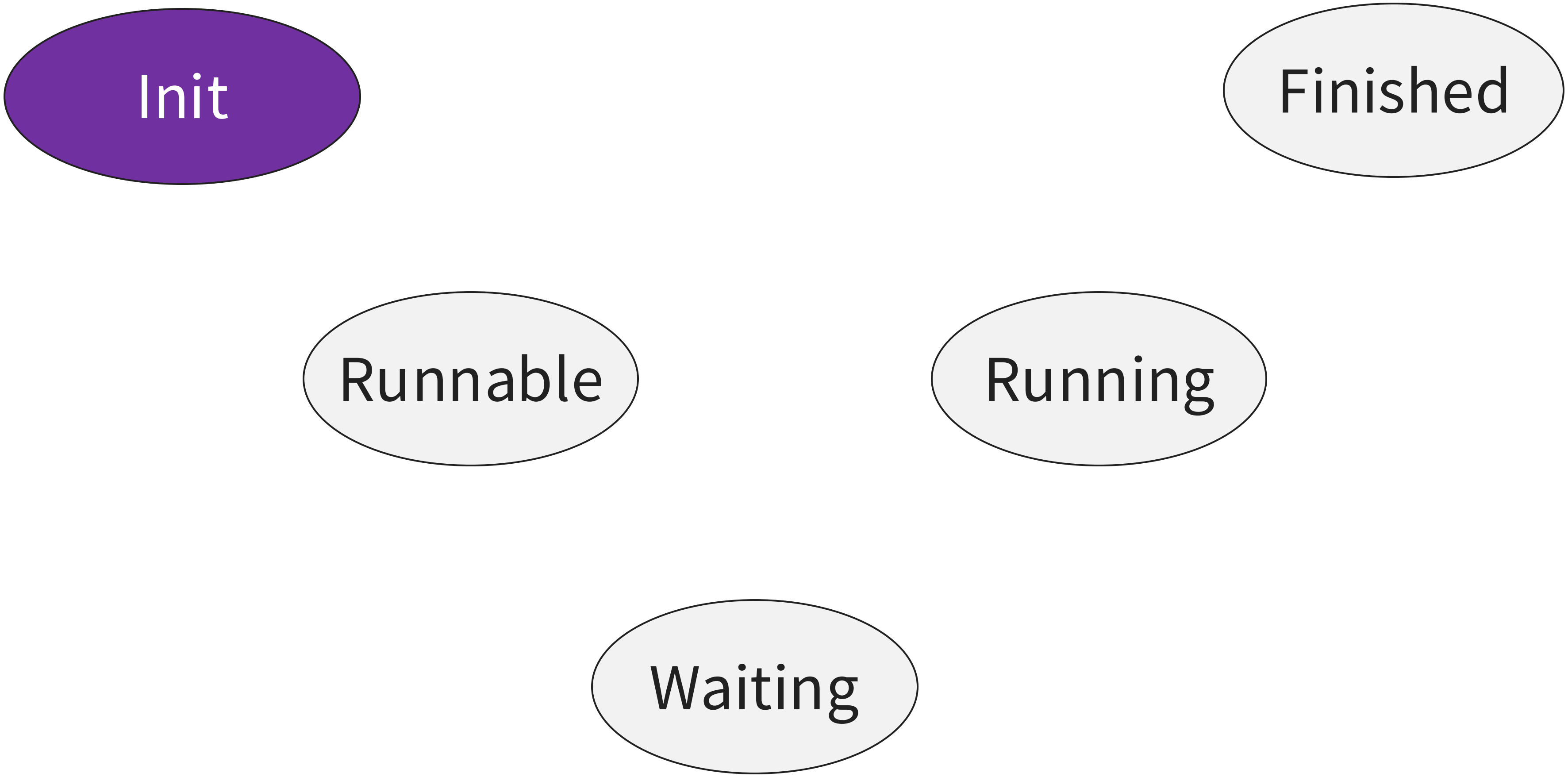
Finished

Runnable

Running

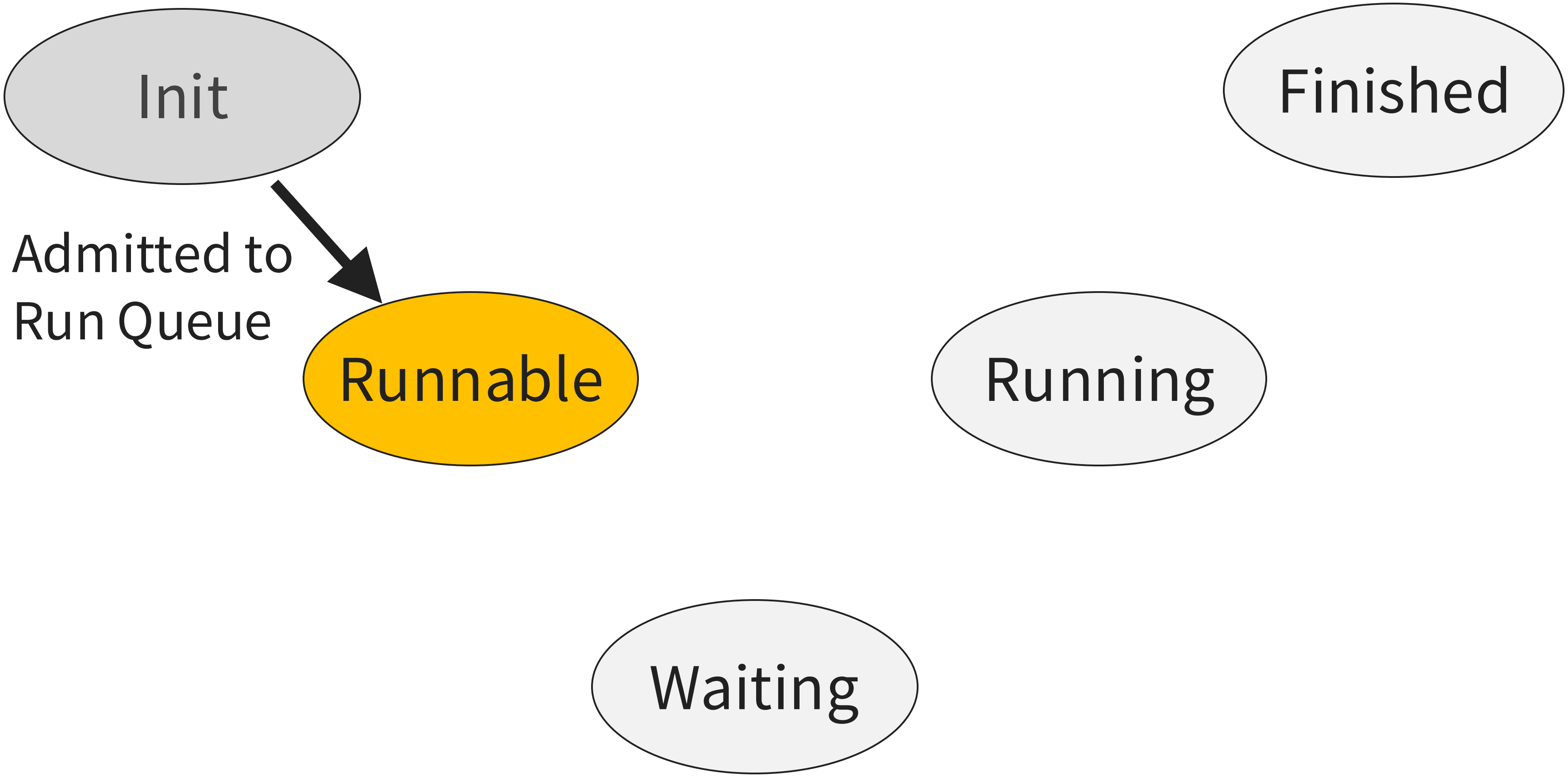
Waiting

Process creation



PCB status: being created
Registers: uninitialized

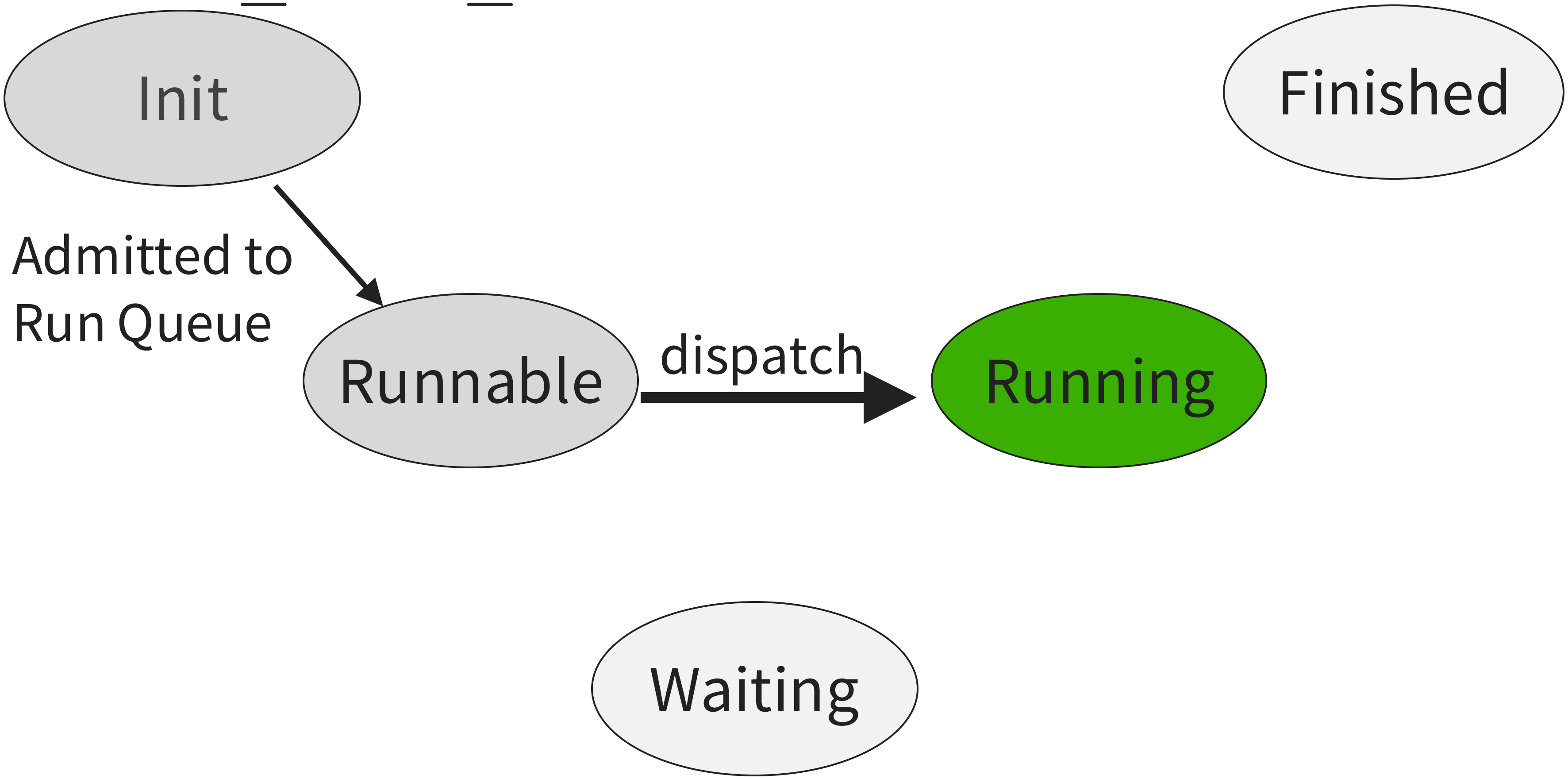
Process is Ready to Run



PCB: on Run Queue (aka Ready Queue)

Registers: pushed by kernel code onto kernel stack

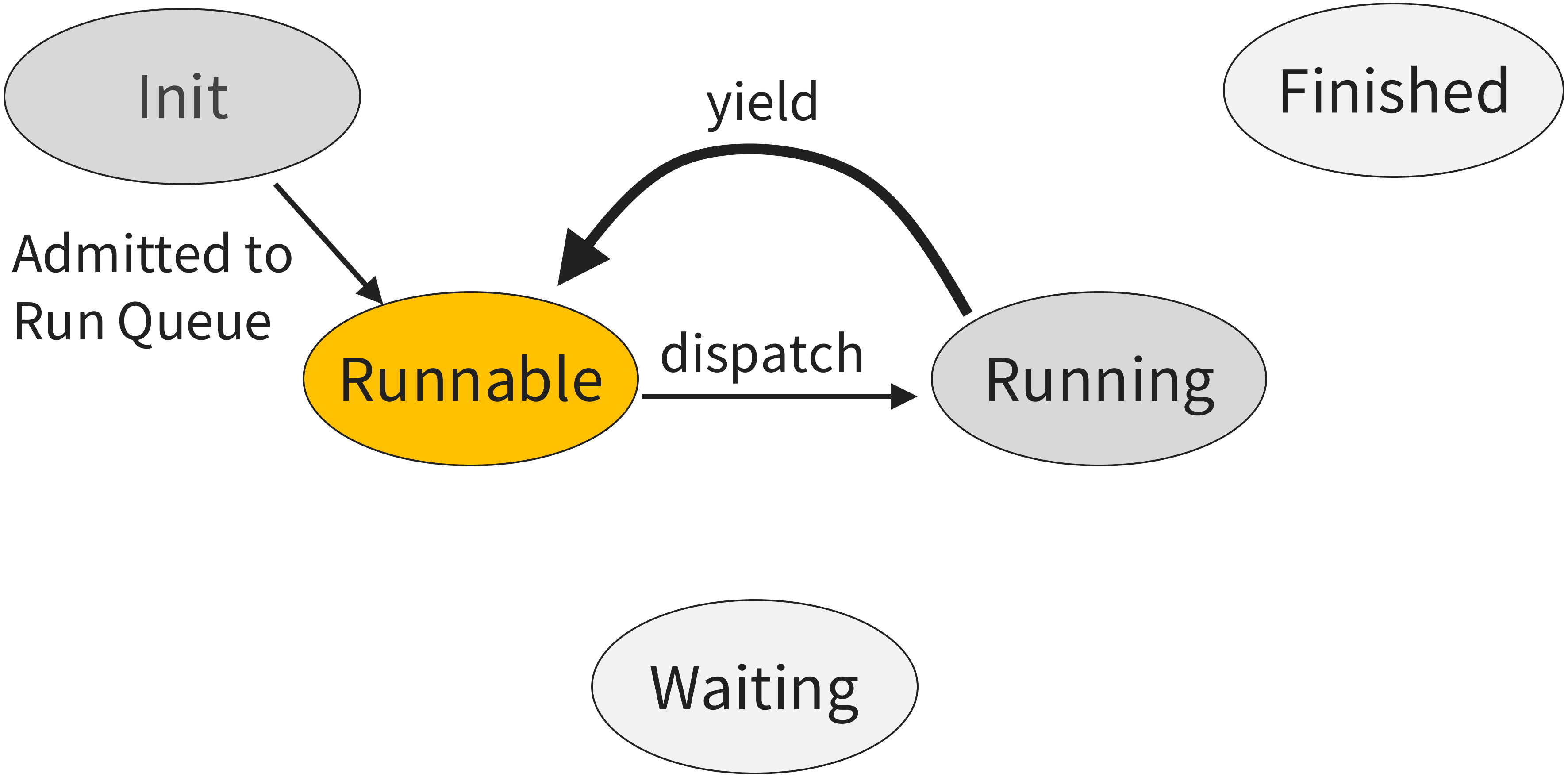
Process is Running
(in supervisor mode, but may `return_from_interrupt` to user mode)



PCB: currently executing

Registers: popped from kernel stack into CPU

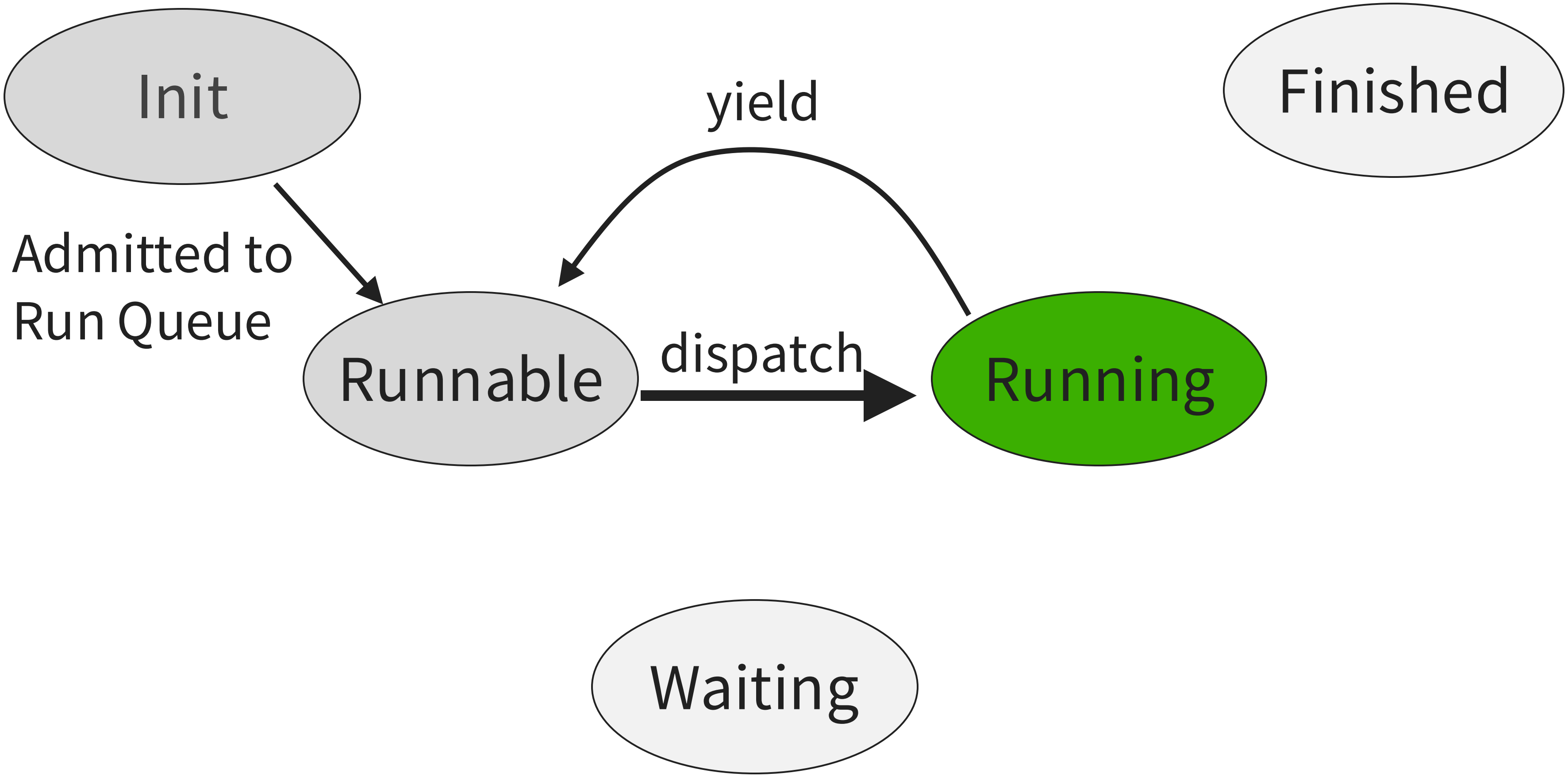
Process Yields (on clock interrupt)



PCB: on Run queue

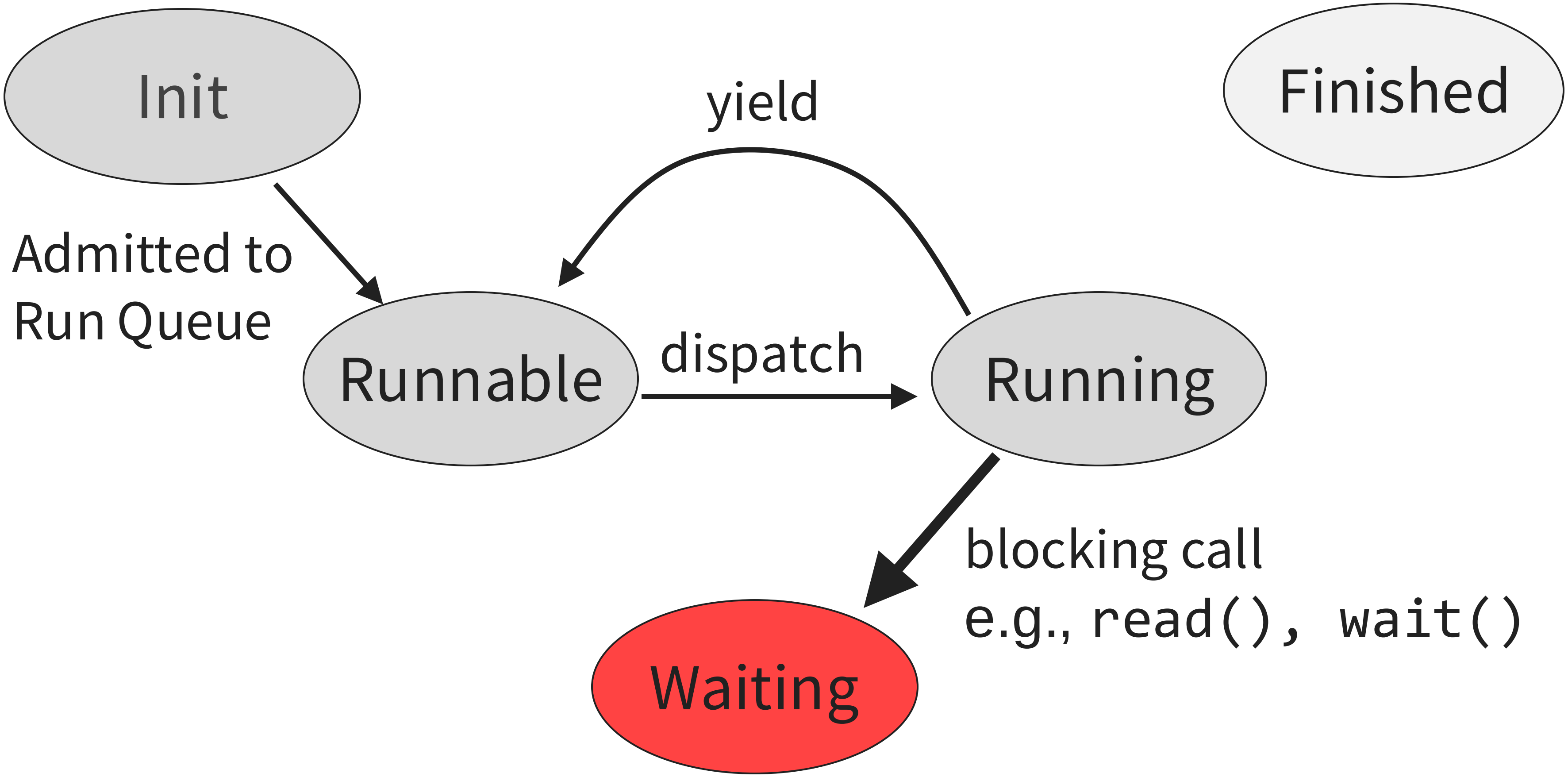
Registers: pushed onto kernel stack (sp saved in PCB)

Process is Running Again!



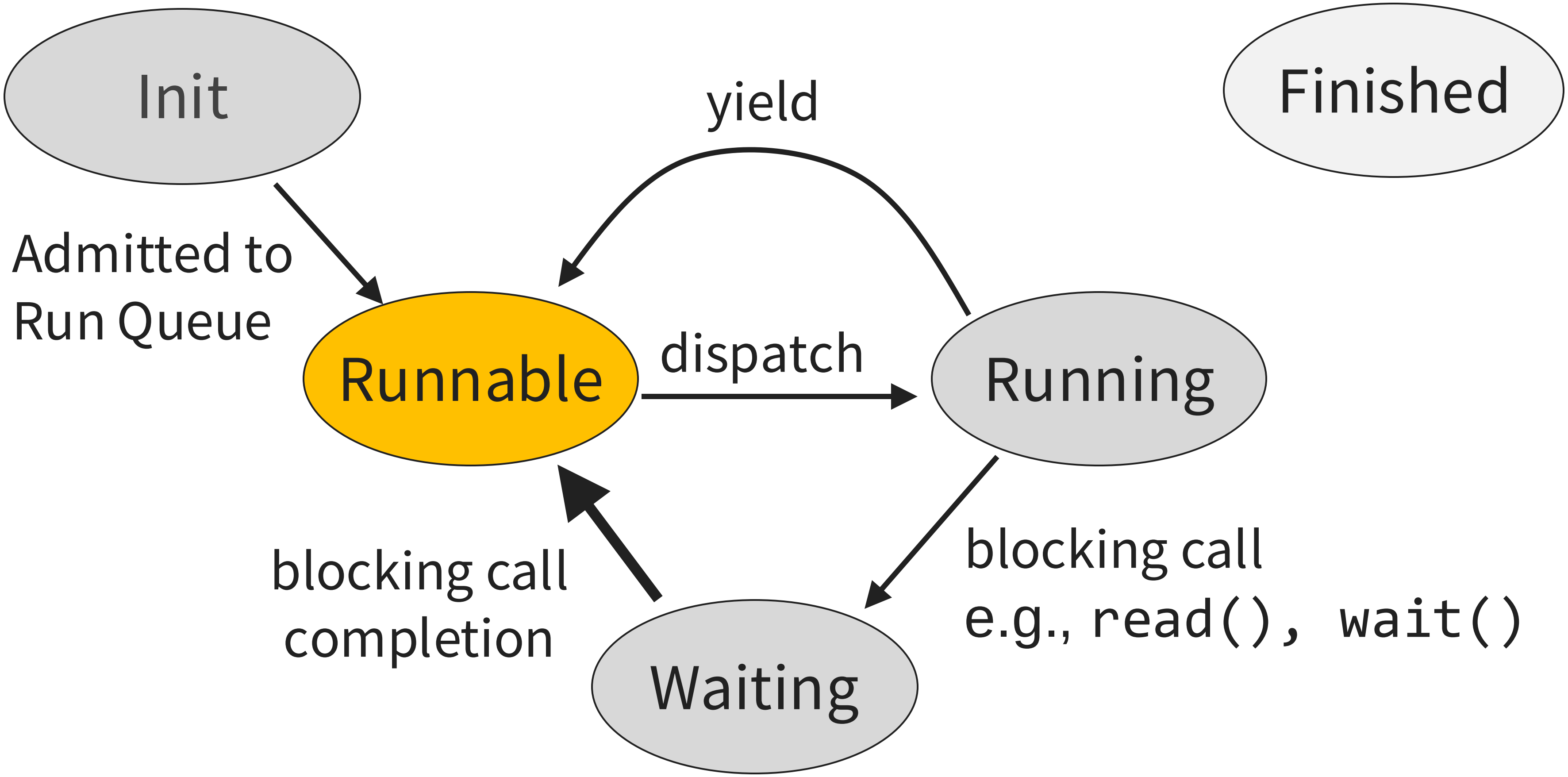
PCB: currently executing
Registers: sp restored from PCB; others restored from stack

Process is Waiting



PCB: on specific waiting queue (file input, …)
Registers: on kernel stack

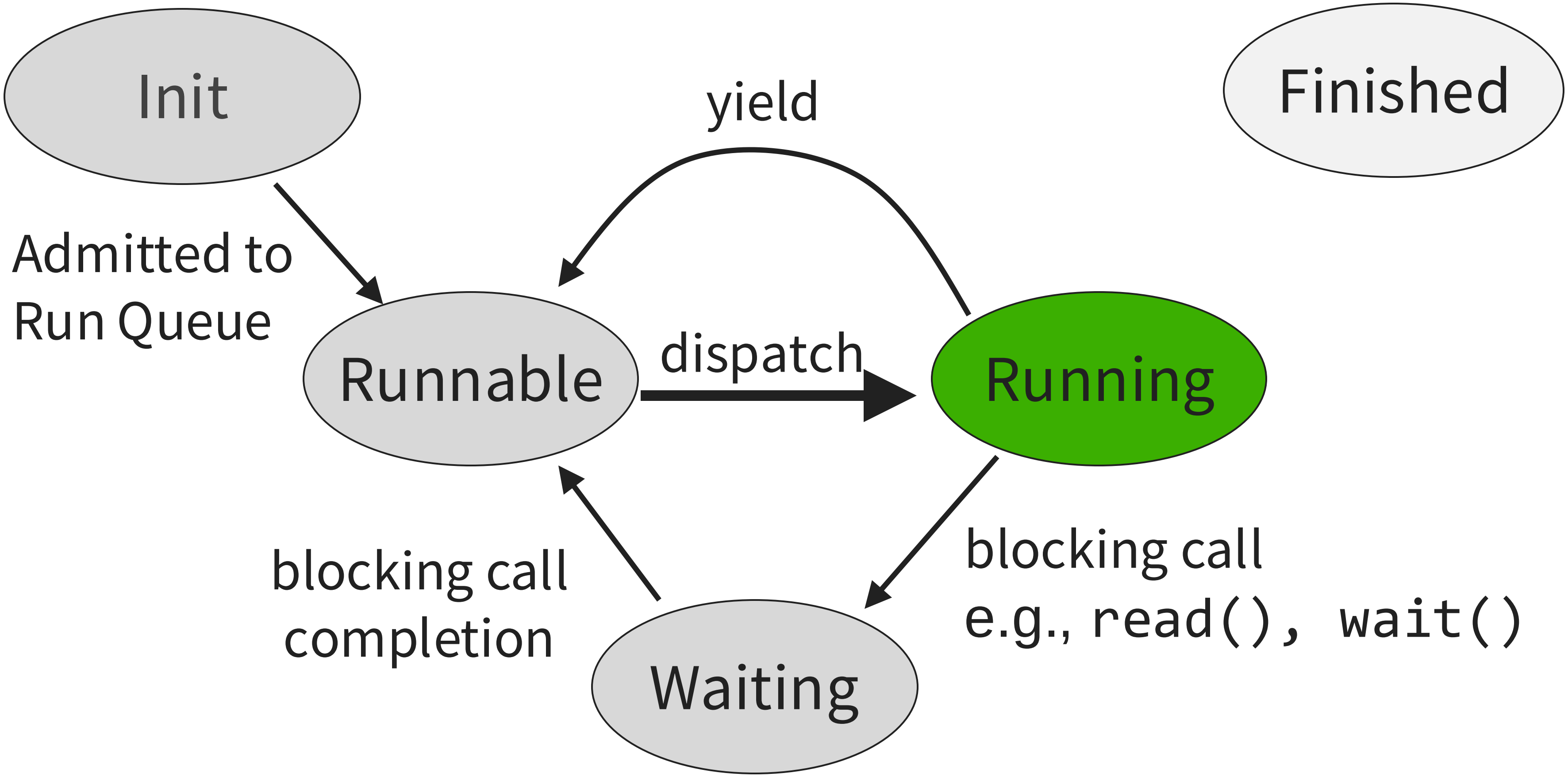
Process is Ready Again!



PCB: on run queue

Registers: on kernel stack

Process is Running Again!



Init

Finished

yield

Admitted to
Run Queue

Runnable

dispatch

Running

blocking call
completion

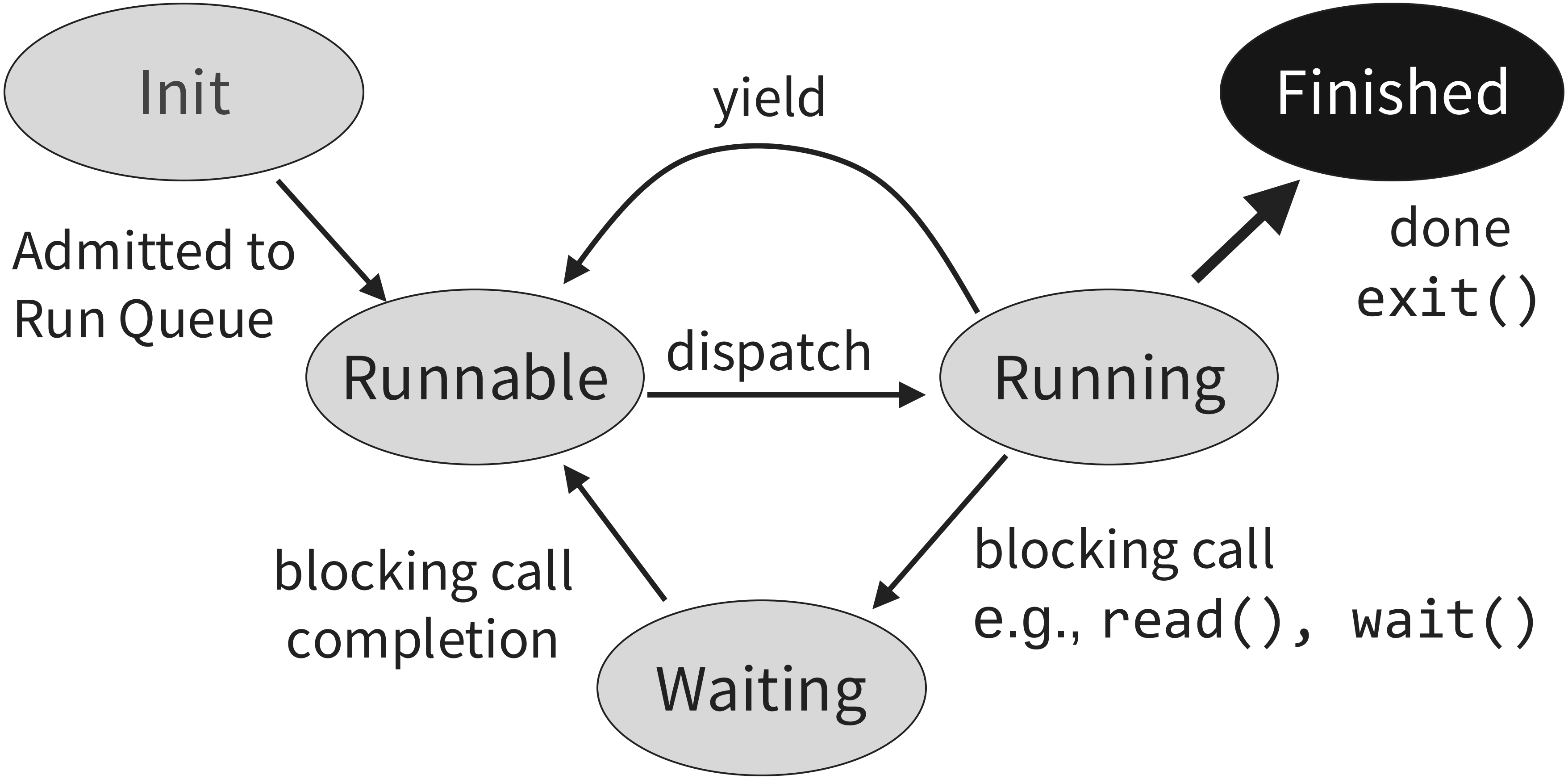
Waiting

blocking call
e.g., `read()`, `wait()`

PCB: currently executing

Registers: restored from kernel 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 kernel stack is empty
- If process is RUNNING
  - its PCB is not on any queue
  - *however, not necessarily in user mode (when servicing interrupt)*
- If process is RUNNABLE or WAITING
  - its kernel 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 never dies and 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 kernel 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 kernel 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;
    assert(current->state == RUNNING);
}
```

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

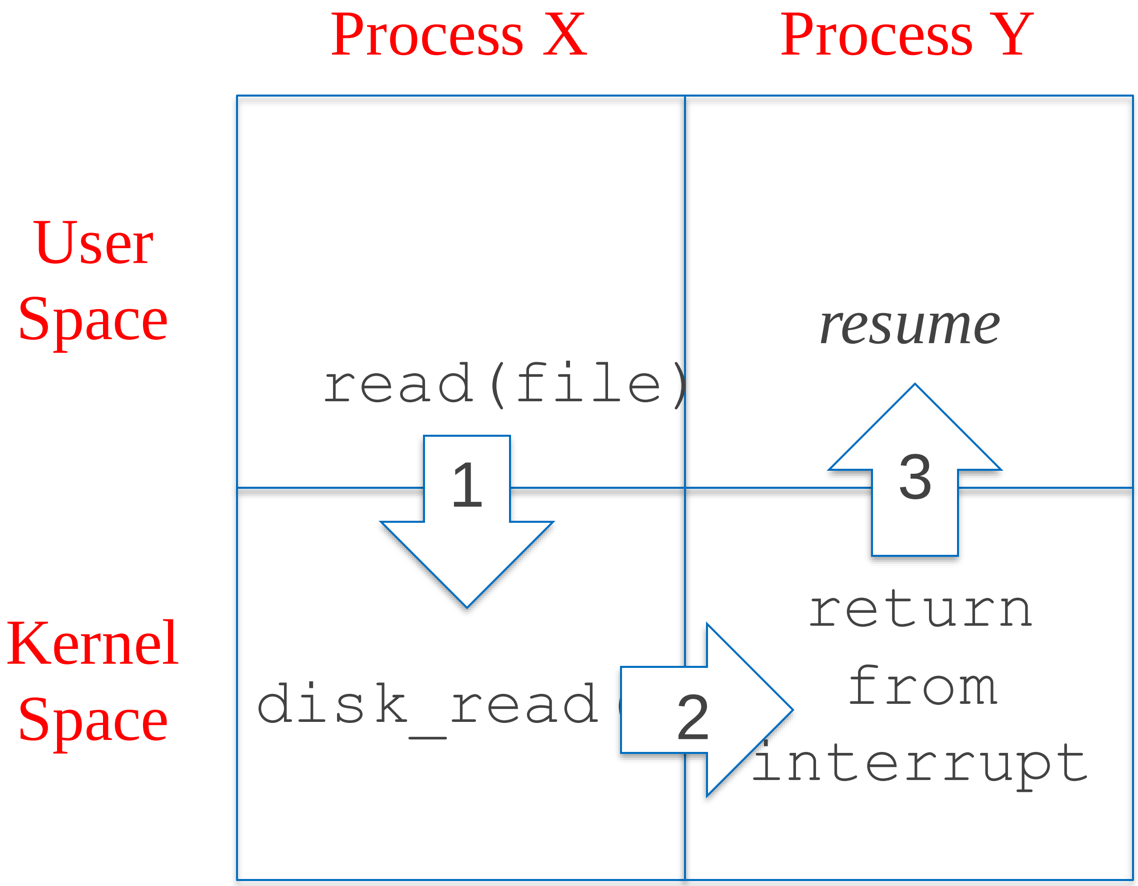
1. Interrupt: From user to kernel space
   - system call, exception, or interrupt
2. Yield: In kernel space, between two processes
   - happens inside the kernel, switching from one PCB/kernel stack to another
3. Return-From-Interrupt: From kernel space to user space
   - Through a `return_from_interrupt`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

Note that each involves a stack switch:

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



Process X

Process Y

User Space

Kernel Space

`read(file)`

resume

1

2

3

`disk_read(`

`return from interrupt`

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
)
```

[Windows]

~~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 )`~~

`)`

[UNIX]

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):

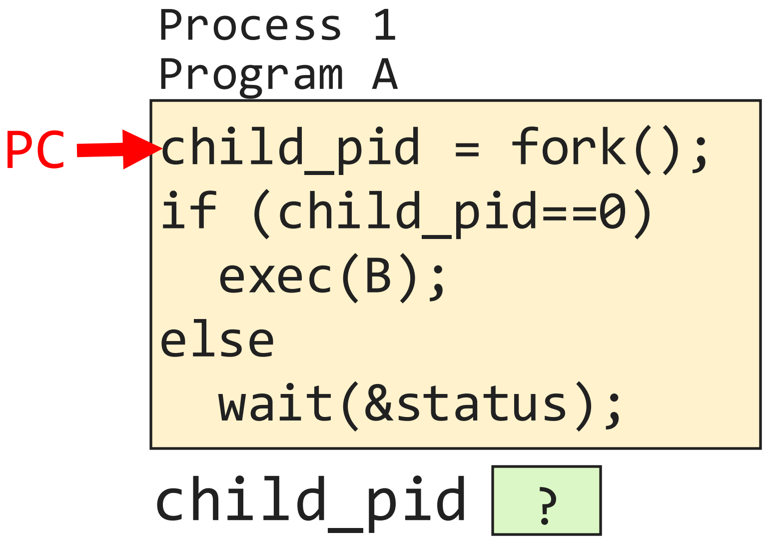
- Load the program into the address space
- Copy arguments into memory in address space
- Initialize h/w context to start execution at “start”

Windows **createProcess(…)** does both

Creating and Managing Processes

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

Fork + Exec



Process 1
Program A

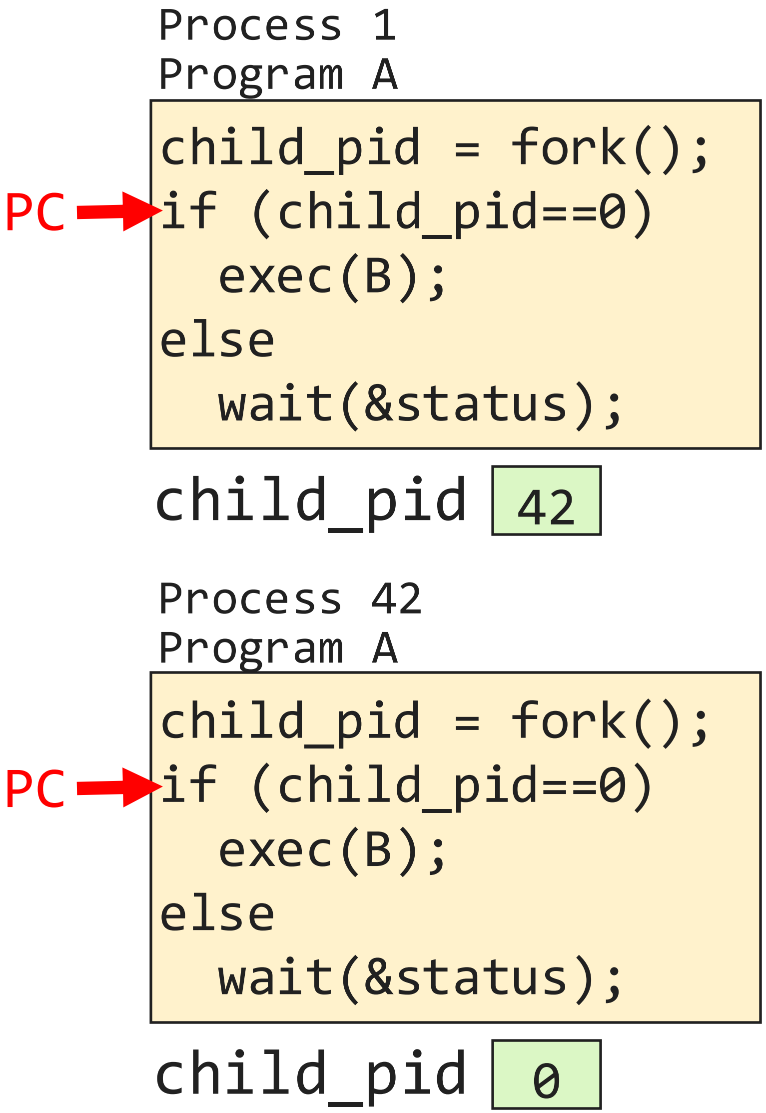
PC

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

child_pid `42`

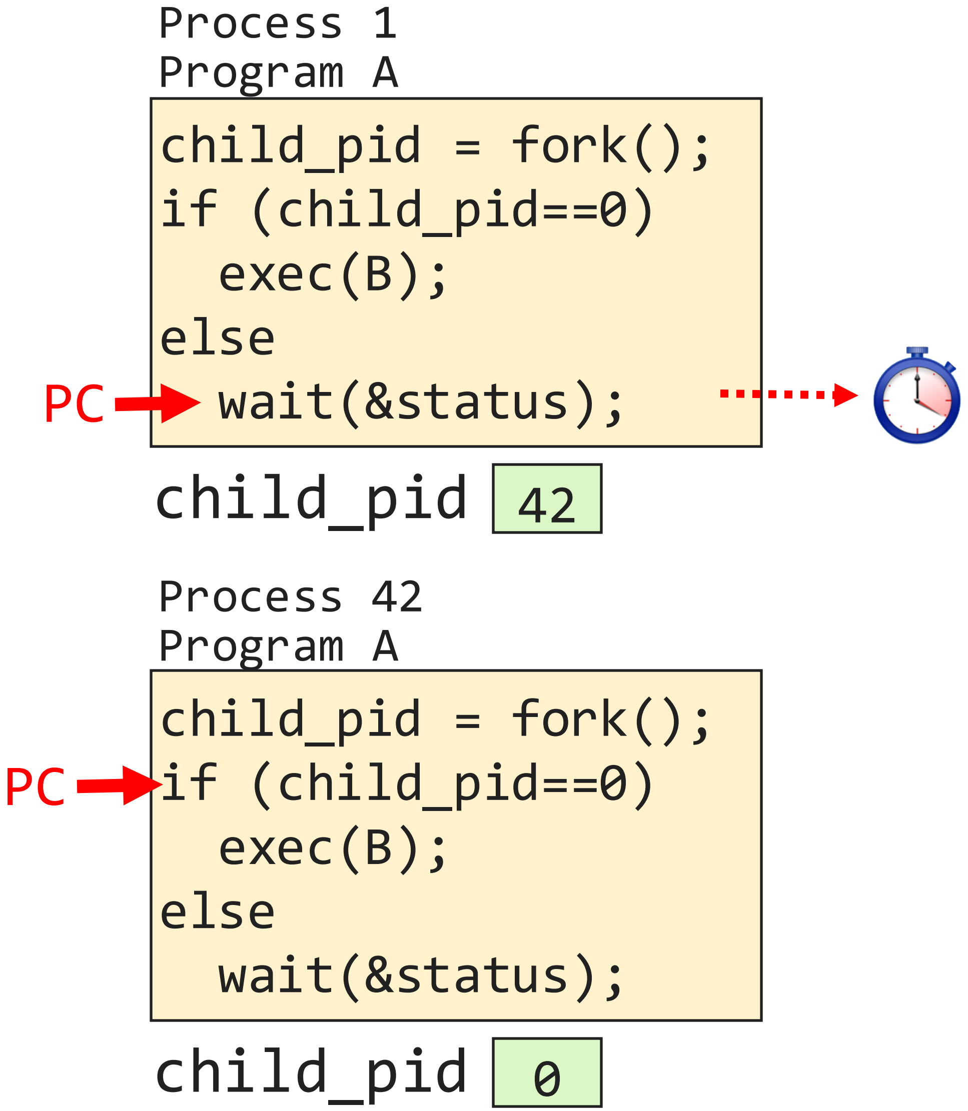
Process 42
Program A

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

child_pid `0`

[UNIX]

Fork + Exec



```
Process 1
Program A
child_pid = fork();
if (child_pid==0)
  exec(B);
else
  wait(&status);
```

child_pid `42`

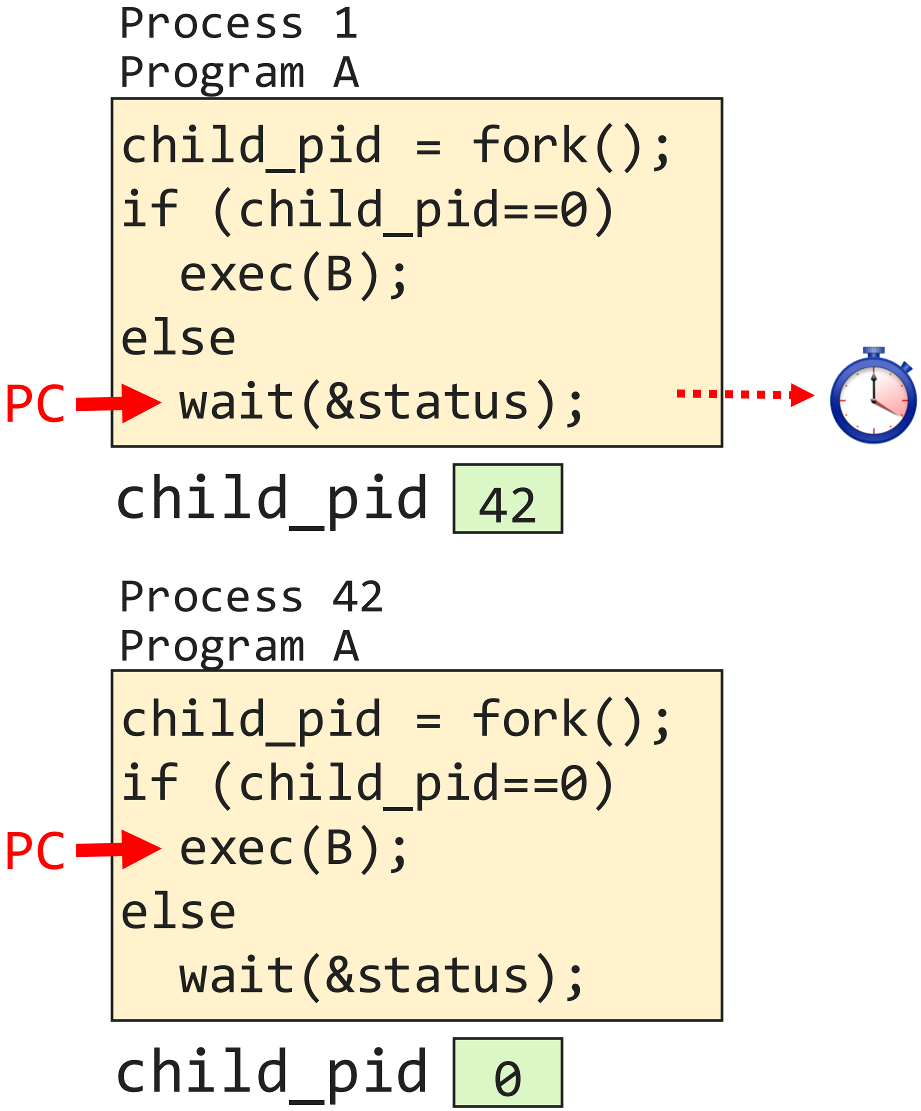
```
Process 42
Program A
child_pid = fork();
if (child_pid==0)
  exec(B);
else
  wait(&status);
```

child_pid `0`

Waits until child exits.

[UNIX]

Fork + Exec



Process 1
Program A

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

child_pid 42

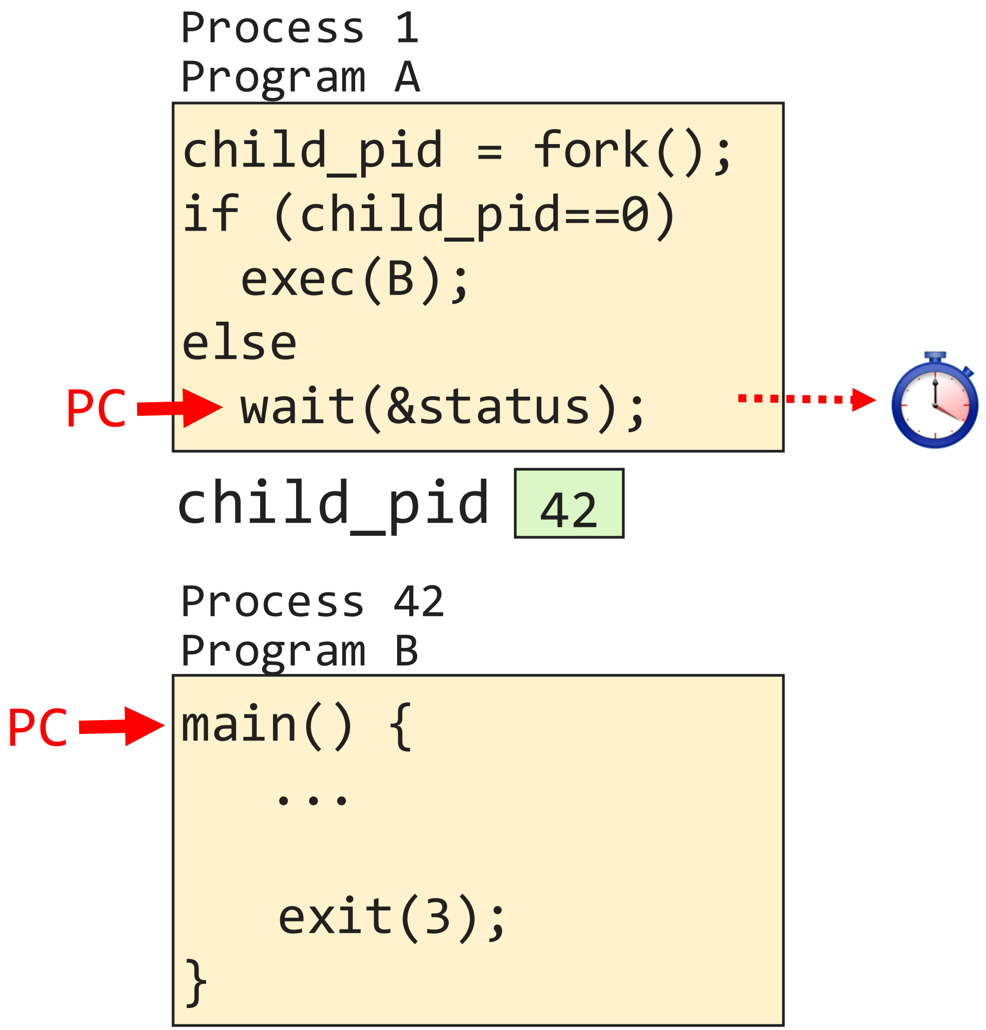
Process 42
Program A

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

child_pid 0

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

Fork + Exec



```
Process 1
Program A
```

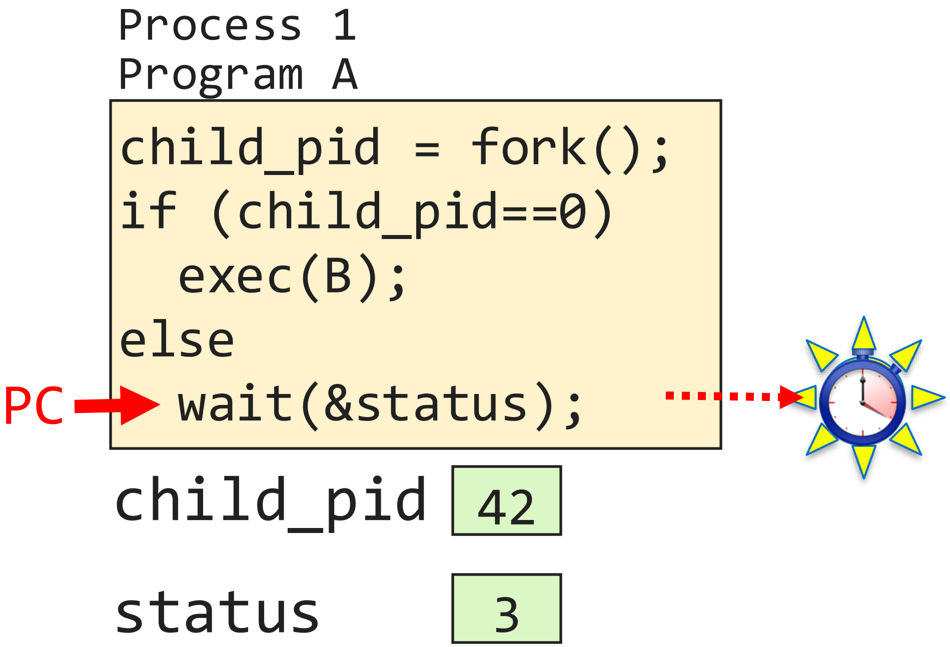
```
child_pid = fork();
if (child_pid==0)
  exec(B);
else
  wait(&status);
```

```
child_pid  42
```

```
Process 42
Program B
```

```
main() {
   ...

   exit(3);
}
```

Fork + Exec



```
Process 1
Program A
```

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

Shell

What is a Shell?

- is an interpreter (i.e., just another program)
- language allows user to create/manage programs
- Example shells:
  - sh Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
  - bash "Bourne-Again" Shell (free, Linux, MacOSX)
  - cmd Windows shell (Therese Stowell, Microsoft, 1987)
  - PowerShell (2006)
  - …

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
- Programs are functions you can call!
- e.g. (sh, bash):

```
$ for student in aa12 klm666 xyz32
> do
      > echo $student             # echo is a print command
      $ if gcc $student/program.c
      > then echo program of $student compiled!
      > else echo program of $student is broken
      > fi
> done
```

What is a Shell?

- Reads lines of input
  - command [arg1 …]
- And executes them
- Full programming language in its ow
- Programs are functions you can call!
- e.g. (sh, bash):

Folder with one subfolder per student

(this is what CMSX gives me)

```
$ for student in `ls Submissions`
> do
      > echo $student              # echo is a print command
      $ if gcc $student/program.c
      > then echo program of $student compiled!
      > else echo program of $student is broken
      > fi
> done
```

“flags” (aka options)

- arguments to command that start with ‘-’
  - this is a convention, not a rule
- examples:
  - ls –l # long listing
  - ps –a # print all processes

Shell has state

- Just like other programming languages
- State includes:
  - environment variables
  - home directory (directory == folder)
  - working directory
  - list of processes
- Commands often modify the state

Environment Variables

- Each process has access to a collection of *environment variables*
  - implicit arguments to the process
- Each env variable has a name and a value
  - both are strings
- One env variable is the search "path"
  - list of folders/directories to find executables
- For example:
  - **PATH**=/bin:/usr/bin:/usr/local/bin
  - export PATH
  - echo $PATH

Some important sh commands

- echo [args] # print arguments
- man cmd # print manual page for cmd
- ls [-l] # list the working directory
- pwd # print working directory
- cd [dir] # change working directory
  - default is “home” directory
- ps [-axl] # list running processes
- kill [-SIG] PID # send signal to process PID
  # signal 9 terminates PID

$*x* evaluates to the value of variable *x*

“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)&`

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`

Threads! (Chapters 25-27)

Other terms for threads:

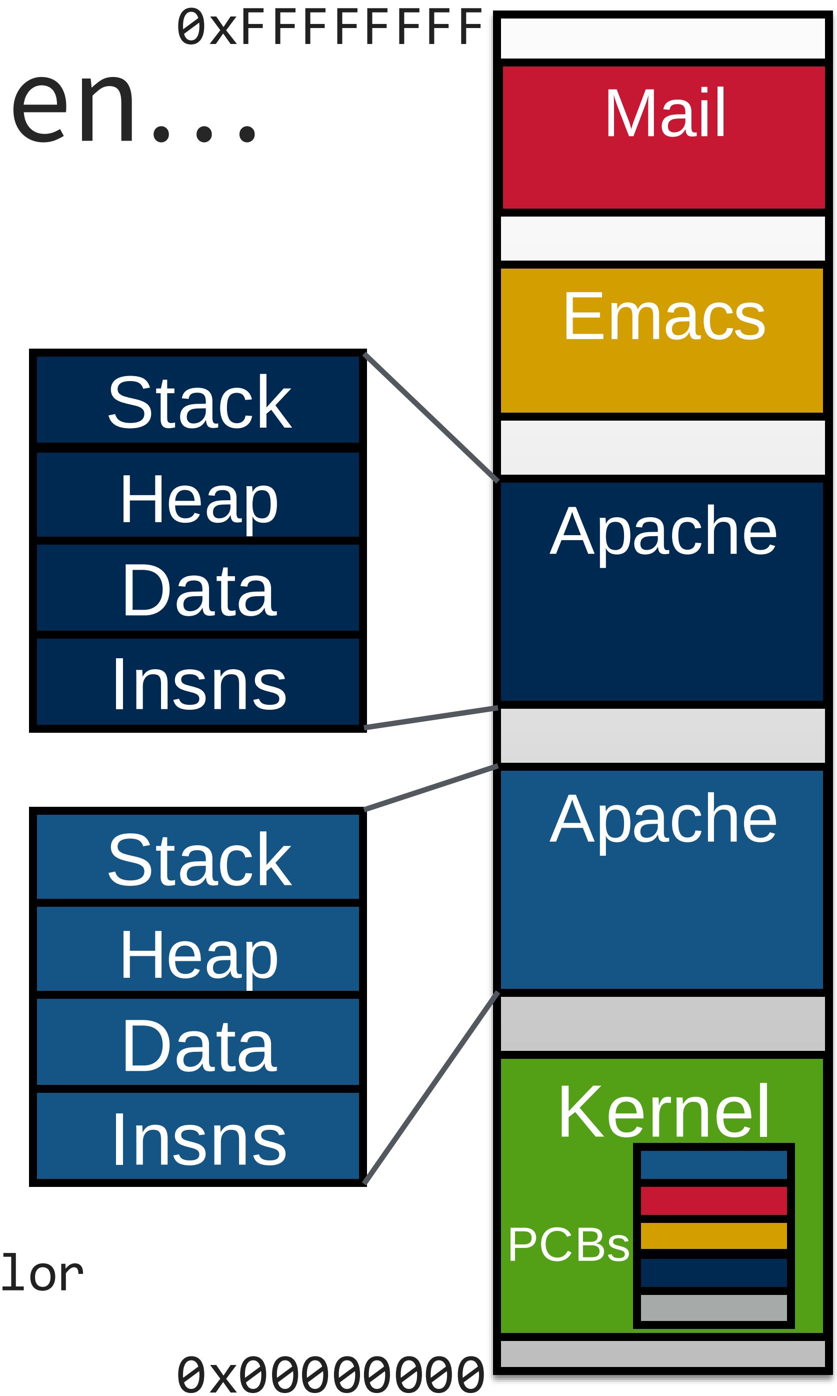
- Lightweight Process
- Thread of Control
- Task

What happens when…

Apache wants to run multiple concurrent computations?

Two heavyweight address spaces for two concurrent computations

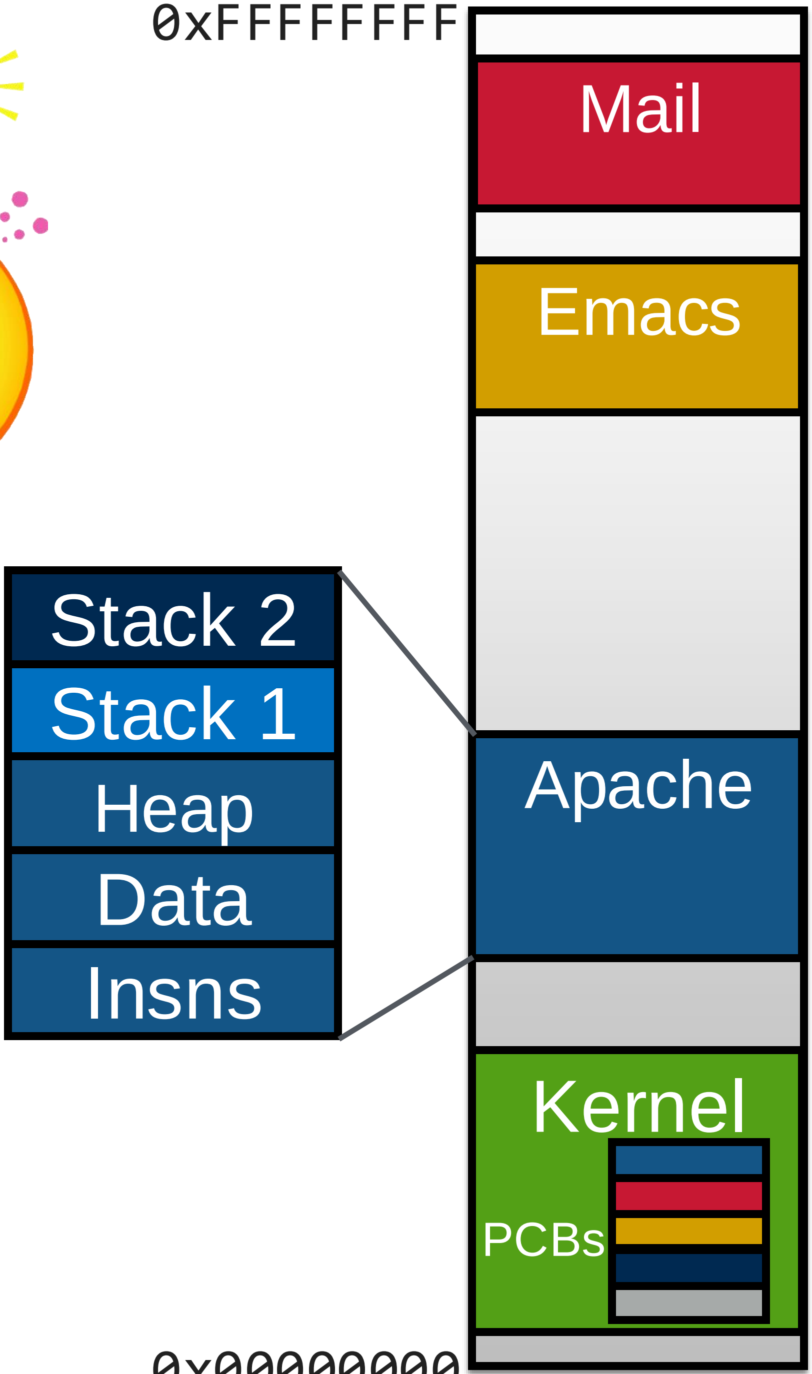
Hard to share cache, etc.



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

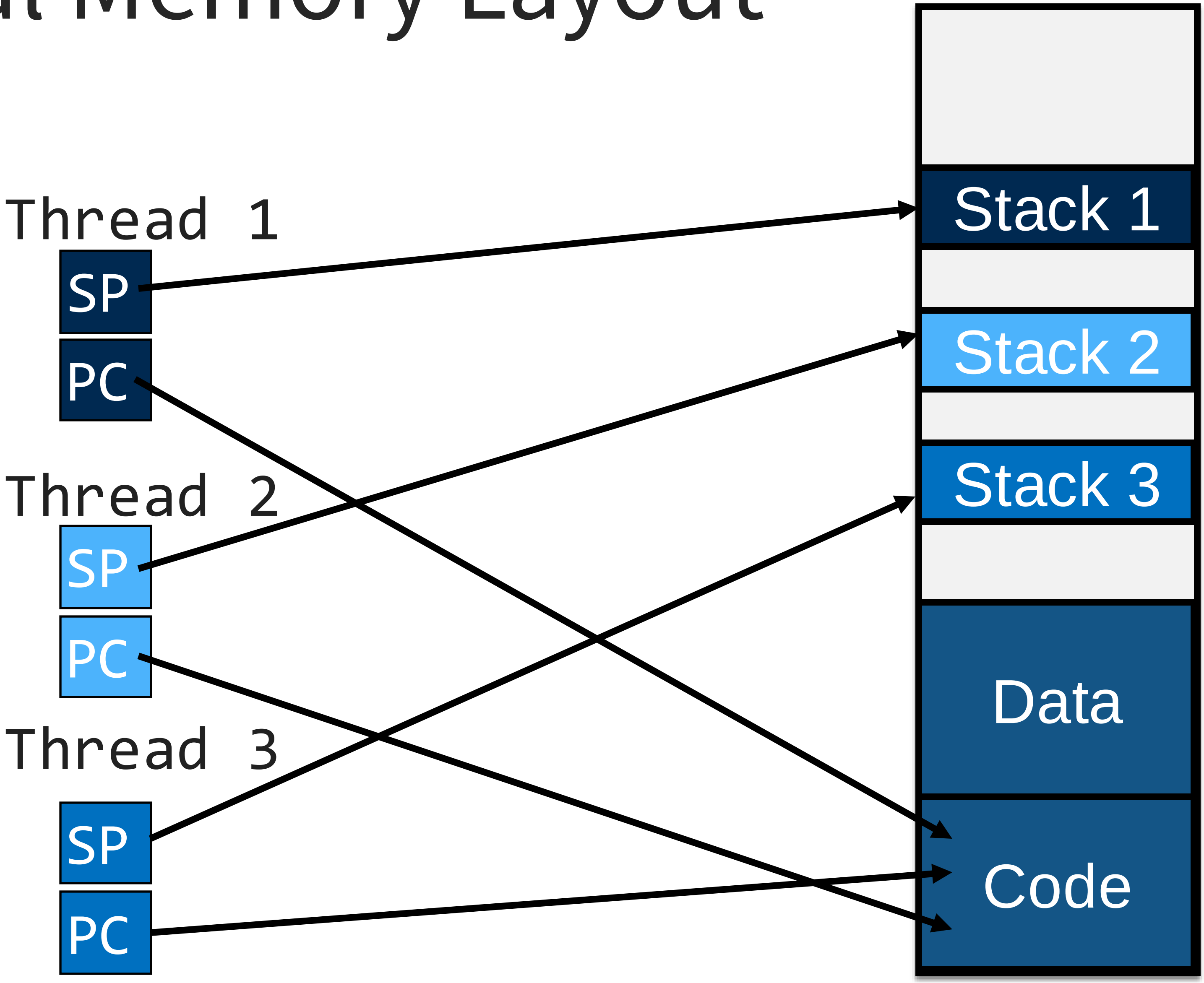
- A process is an abstraction of a computer
  - CPU, memory, devices
- A thread is an abstraction of a core
  - registers (incl. PC and SP)

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?

Virtual Memory Layout



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

Two Thread Examples

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

Web server thread:

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

Simple Thread API

`void` **`thread_create`** `(func,arg)`	Creates a new thread that will execute function **func** with the arguments **arg**
`void` **`thread_yield()`**	Calling thread gives up processor. Scheduler can resume running this thread at any point.
`void` **`thread_exit()`**	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

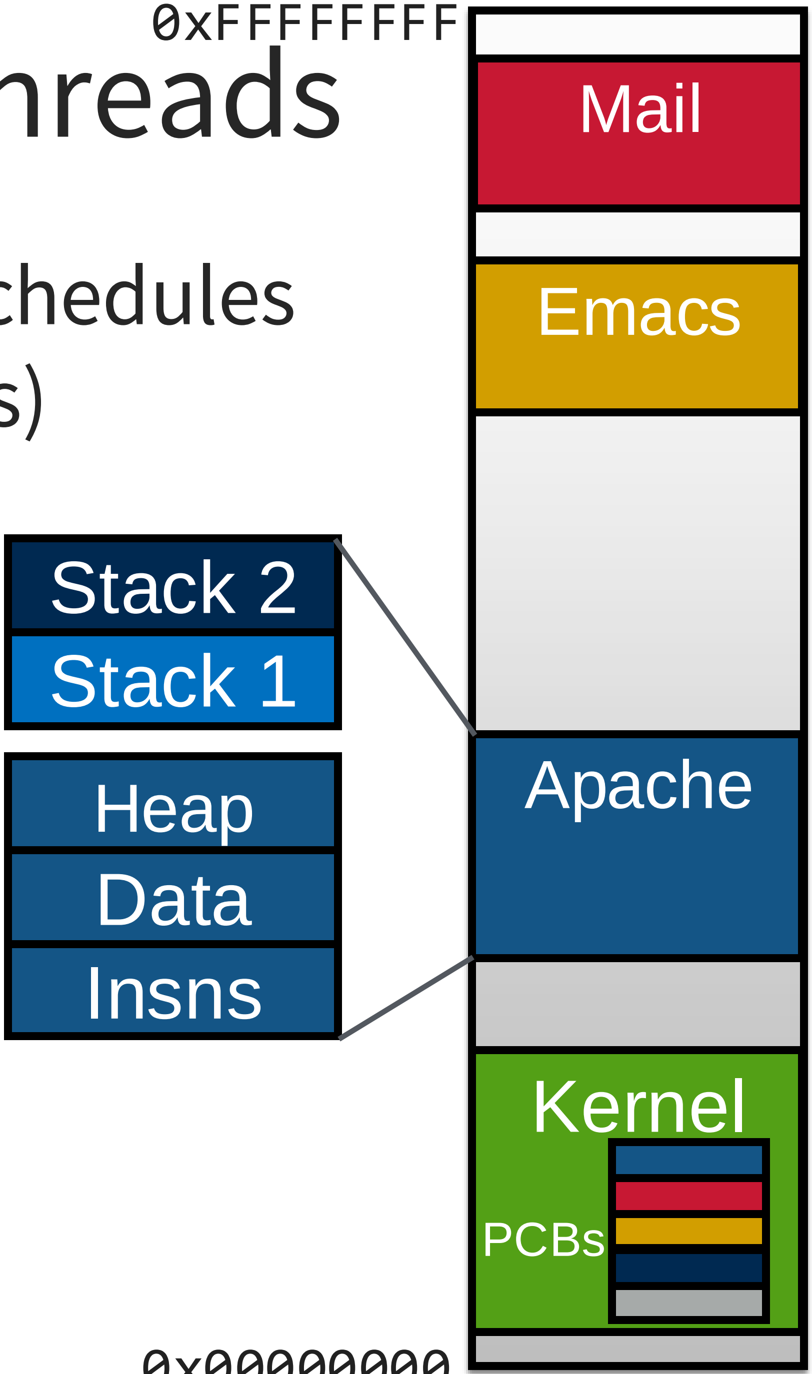
One abstraction, two implementations:

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 and schedules threads (just like processes)

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

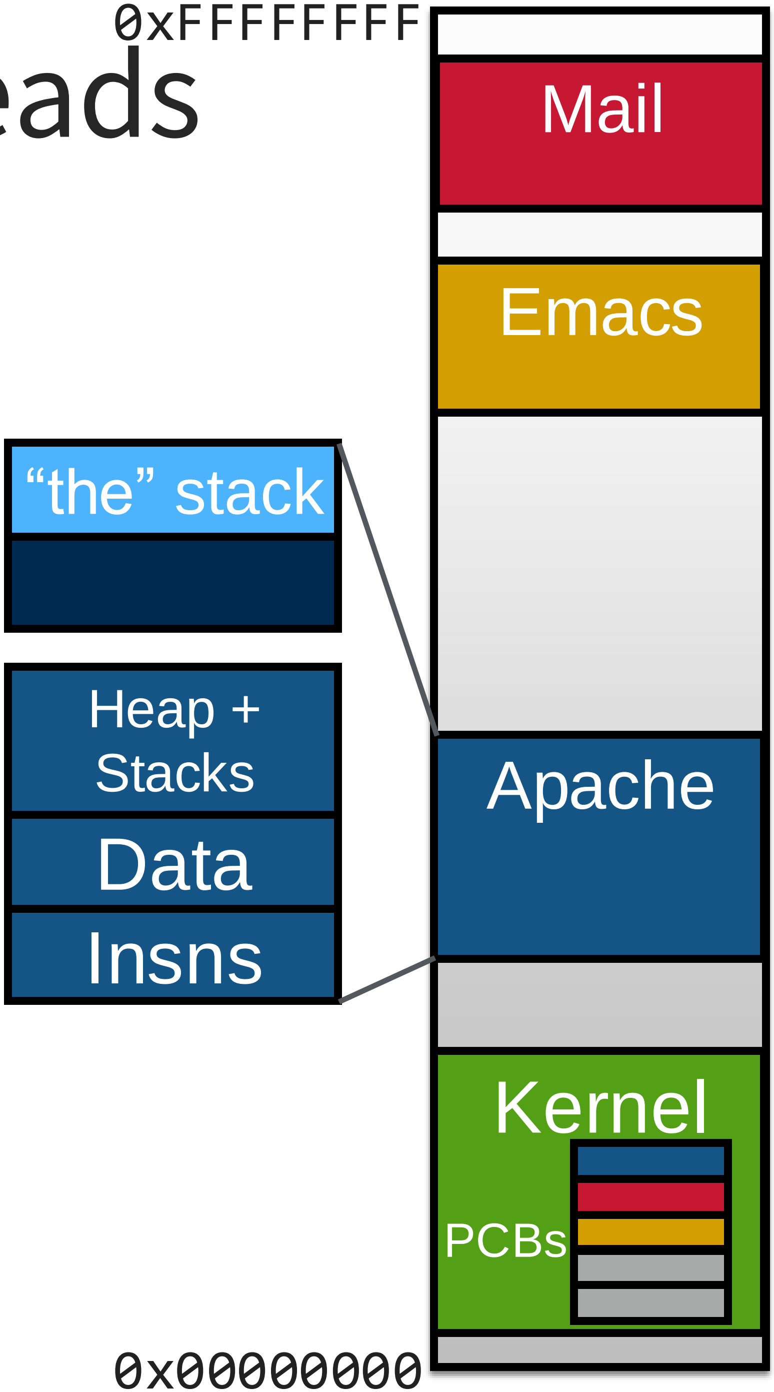


#2: User-Level Threads

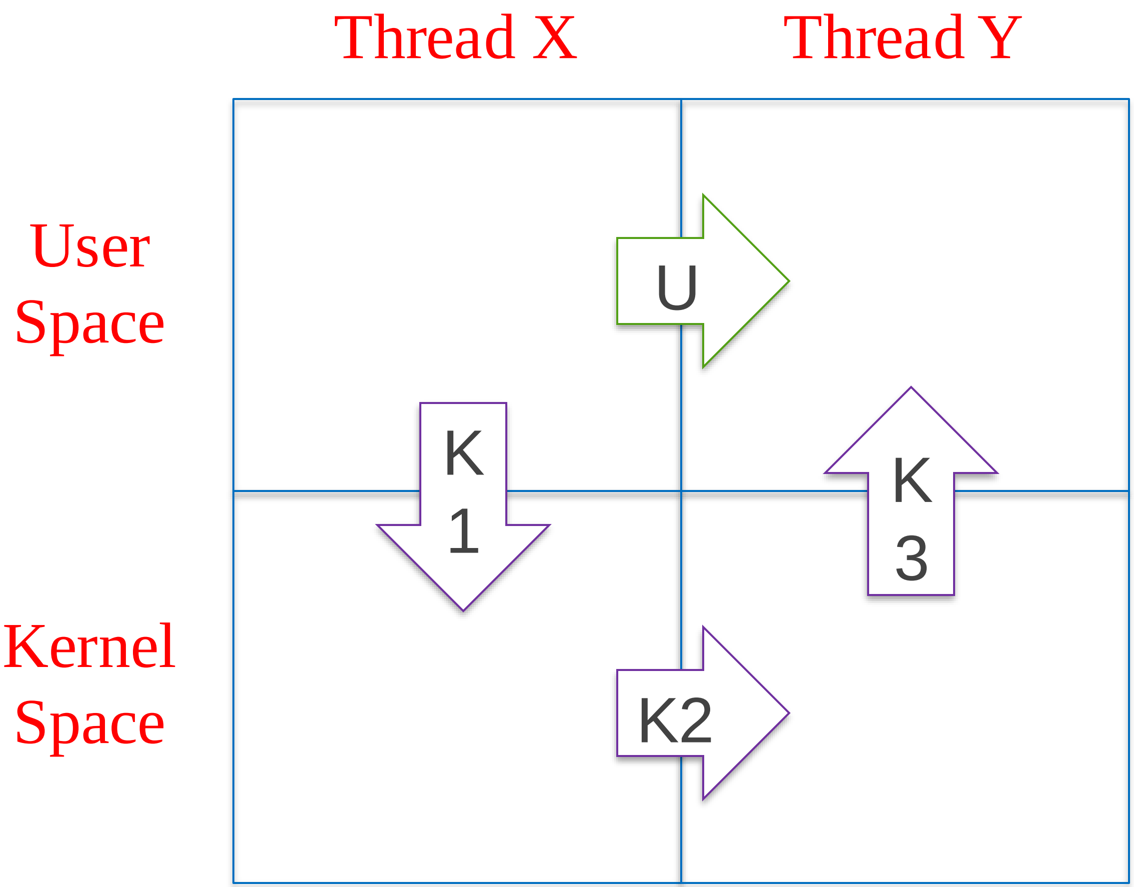
Run mini-OS in user space
- Real OS unaware of threads
- Single PCB
- Thread Control Block (TCB) for each thread

Usually more efficient than kernel-level threads (Why? See next slide)

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



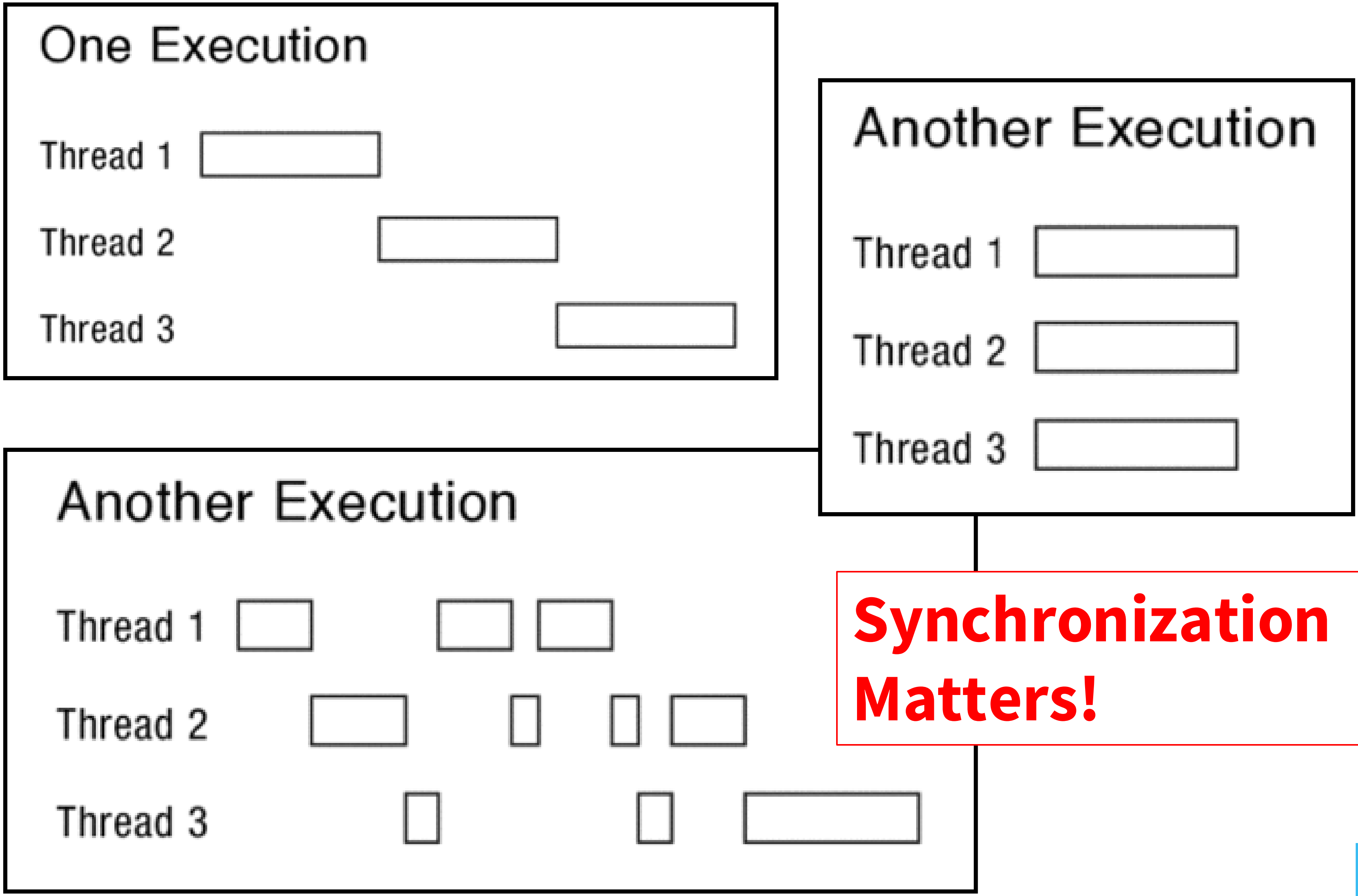
Kernel vs User Thread Switch



Kernel- vs User-level Threads

Kernel-Level Threads	User-level Threads
- Easy to implement: just processes with shared page table	- Requires user-level context switches, scheduler
- Threads can run blocking system calls concurrently	- Blocking system call blocks all threads: needs O.S. support for non-blocking system calls
- Thread switch requires three context switches	- Thread switch efficiently implemented in user space

Do **not** presume to know the schedule



Synchronization Matters!