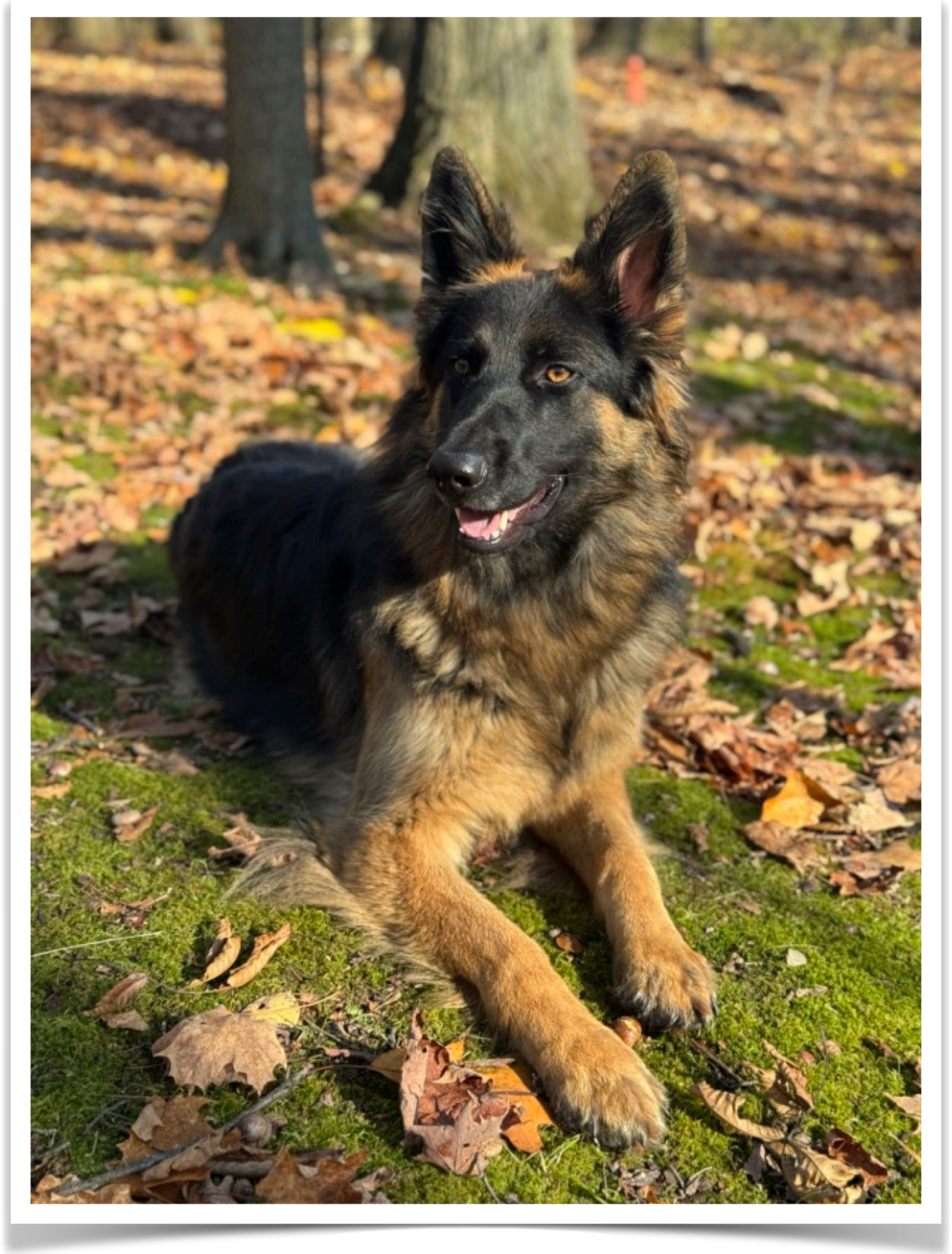
Cornell Bowers CIS
Computer Science



CS3410: Computer Systems and Organization

LEC20: System Calls (Vol. II)

Professor Giulia Guidi
Wednesday, November 5, 2025

Credits: Bala, Bracy, Garcia, Guidi, Kao, Sampson, Sirer, Weatherspoon

Plan for Today

- Review of **processes** and **syscall**
- A bit more on **syscall**
- Brief introduction to virtual memory

Review of OS and processes

If we can run instructions directly on the CPU, why do we need an operating system?

(a) how do multiple programs share CPU and memory without stepping on each other?
(b) how does the OS decide which process gets cache, memory, or I/O?

Process versus Program

recipe

- A **program** consists of code and data
    - It is specified in some programming language, e.g., C
    - It is typically stored in a file on disk
- "Running a program" means creating a **process**
    - **Can run a program multiple times!**
        - One after the other, or even <u>concurrently</u>

person actively cooking from that recipe (ingredients, tools, stove all in use)

A Day in the Life of a Process

The source file: **sum.c** ----------→ The executable: **sum** ------------→ Process is alive: **process** id `pid xxx`

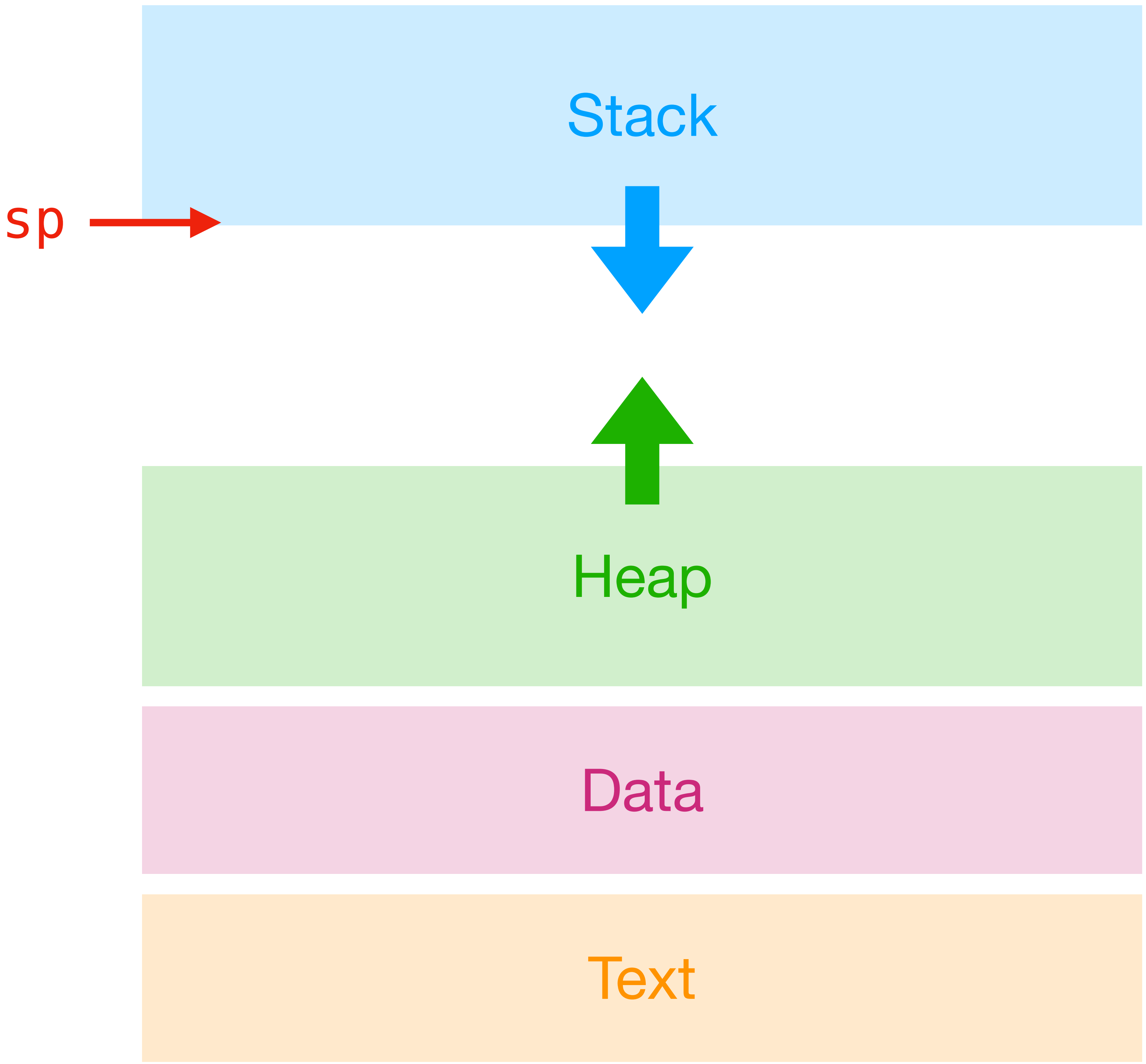
```c
#include <stdio.h>

int max = 10;

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

}
```

program

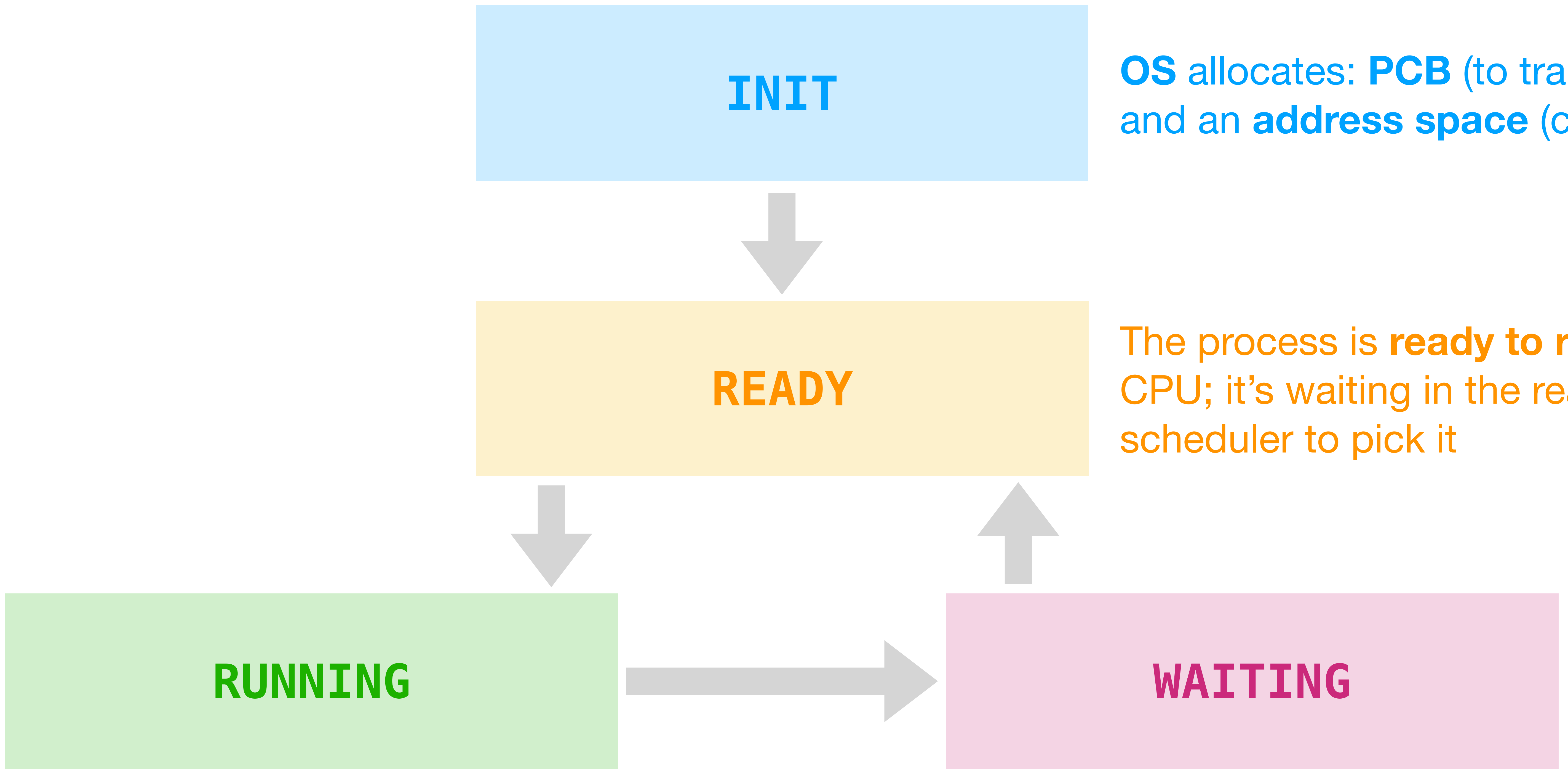
0040 0000 .text main

1000 0000 .data max

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



Process Life Cycle



OS allocates: **PCB** (to track its state, registers, etc.) and an **address space** (code, data, stack)

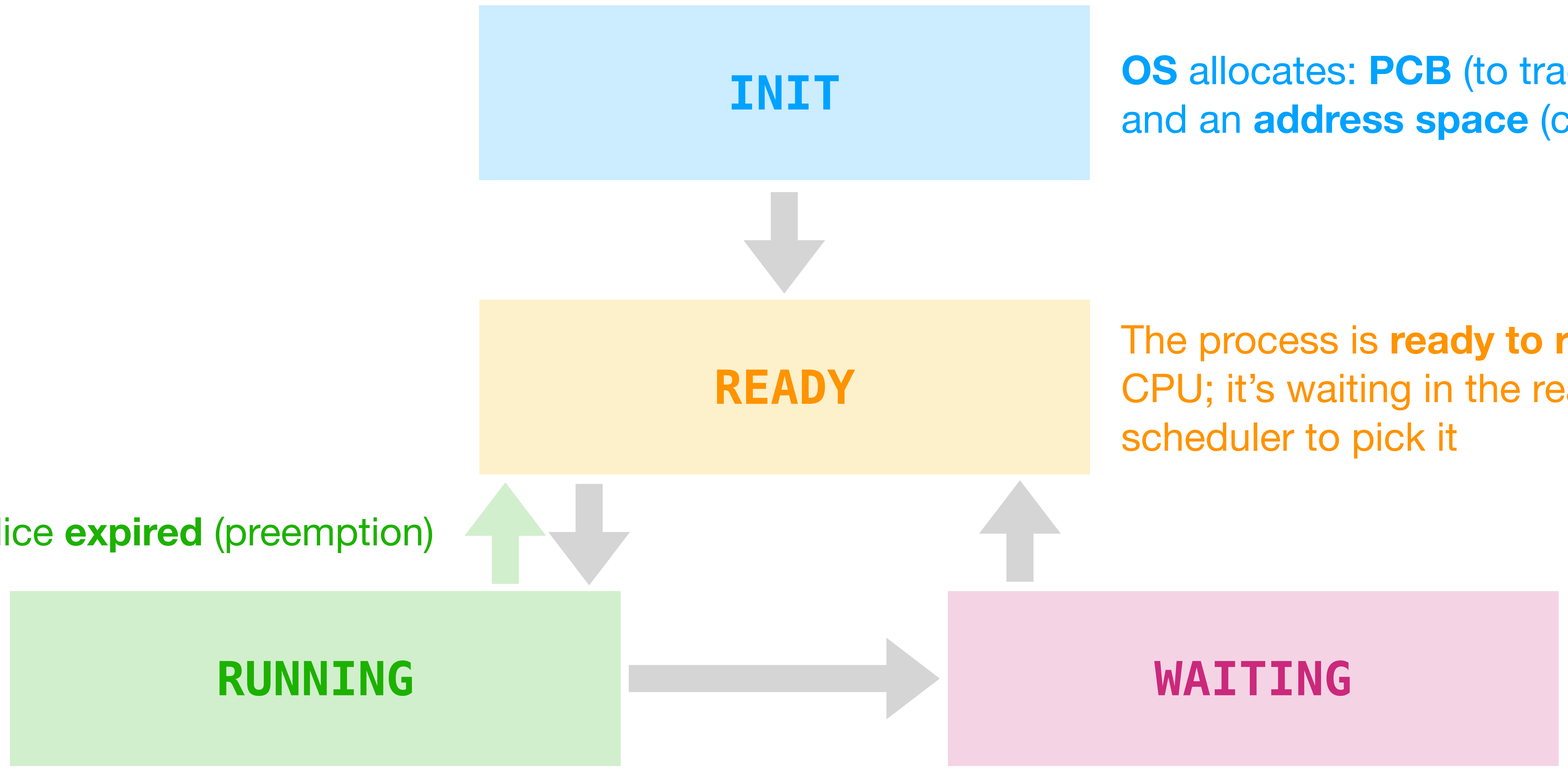
The process is **ready to run**, but **not** currently on the CPU; it's waiting in the ready **queue** for the scheduler to pick it

The process is **now executing** on the CPU

The process **cannot** proceed until some event occurs, e.g., waiting for I/O, waiting for another process (e.g., `waitpid`)

Critical: when the event completes, the OS moves it back to **`READY`**

Process Life Cycle



OS allocates: **PCB** (to track its state, registers, etc.) and an **address space** (code, data, stack)

The process is **ready to run**, but **not** currently on the CPU; it's waiting in the ready **queue** for the scheduler to pick it

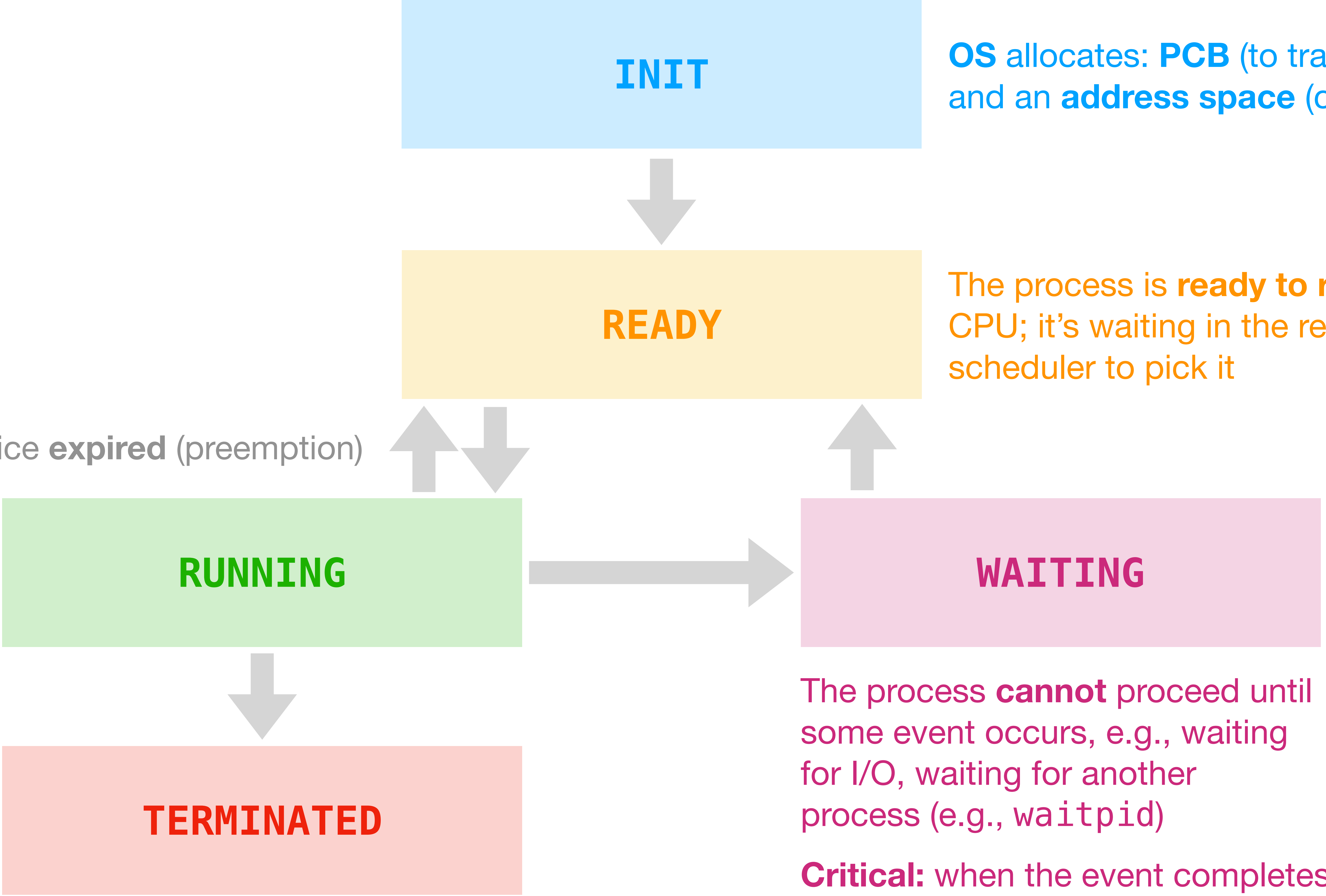
The time slice **expired** (preemption)

The process is **now executing** on the CPU

The process **cannot** proceed until some event occurs, e.g., waiting for I/O, waiting for another process (e.g., `waitpid`)

Critical: when the event completes, the OS moves it back to **`READY`**

Process Life Cycle



OS allocates: **PCB** (to track its state, registers, etc.) and an **address space** (code, data, stack)

The process is **ready to run**, but **not** currently on the CPU; it's waiting in the ready **queue** for the scheduler to pick it

The process is **now executing** on the CPU

The process **cannot** proceed until some event occurs, e.g., waiting for I/O, waiting for another process (e.g., `waitpid`)

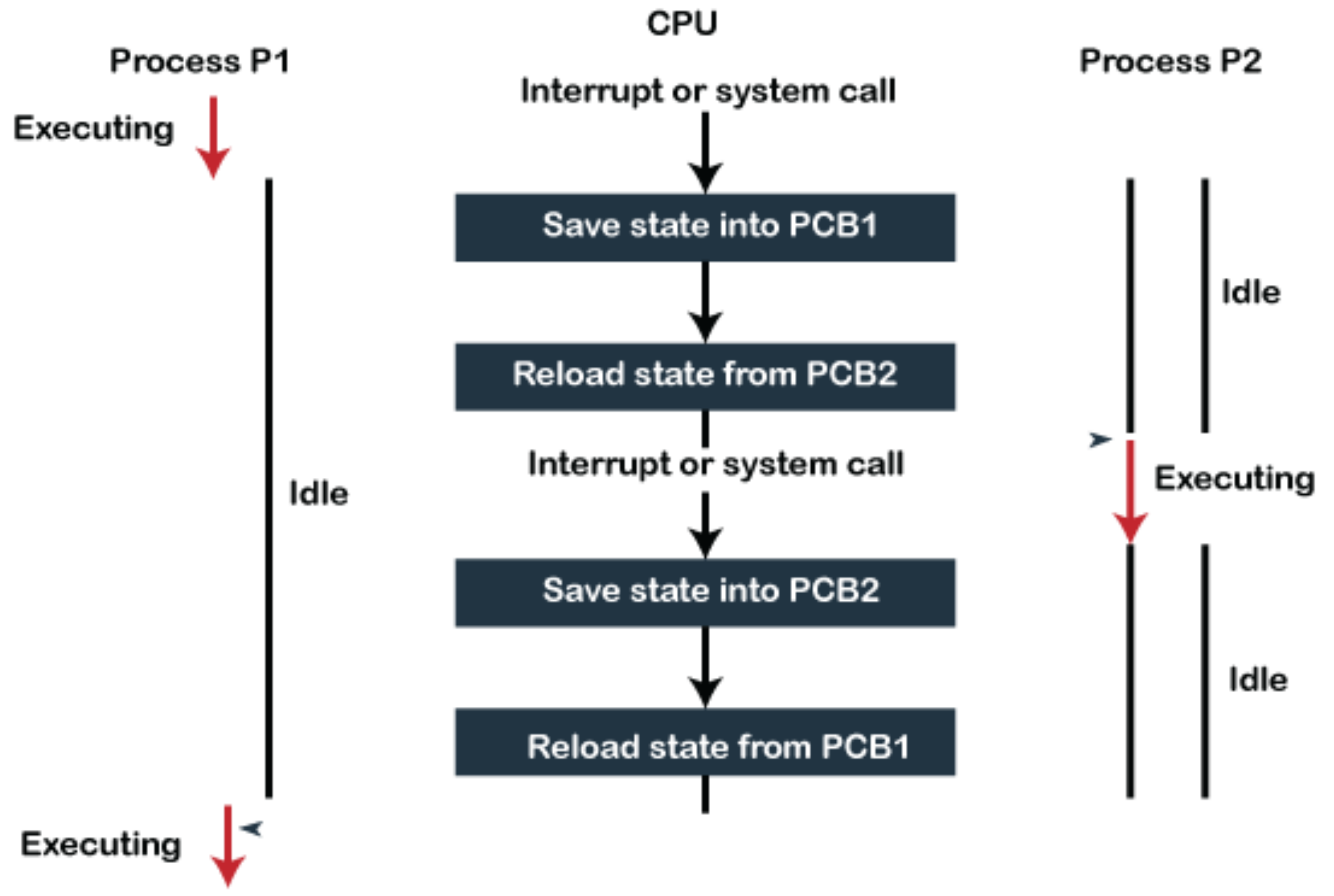
Critical: when the event completes, the OS moves it back to **`READY`**

The process has **finished** executing, e.g., reached `exit()`, frees PCB

Context Switching

The process by which an OS saves the state of a currently running process and restores the state of another process

- First, save the current process state
- Update the Process Control Block (PCB)
- Then, select the next process
- Restore the next process state
- Resume execution



Poll

In an operating system, which statement correctly describes the difference between user space and kernel space?

[PollEv.com /gguidi](PollEv.com/gguidi)

Or send **gguidi** to **22333**

this is where the core of the operating systems (the kernel) runs

User space versus Kernel space

this is where regular programs live (apps, compilers, browsers, your code, etc.)

User Space versus Kernel Space

- **User space** is where programs (apps, compilers, browsers, your code, etc.) run
  - User space applications **cannot** directly access the system's hardware resources
  - It is **restricted** and **isolated** from the kernel space to ensure system stability and security

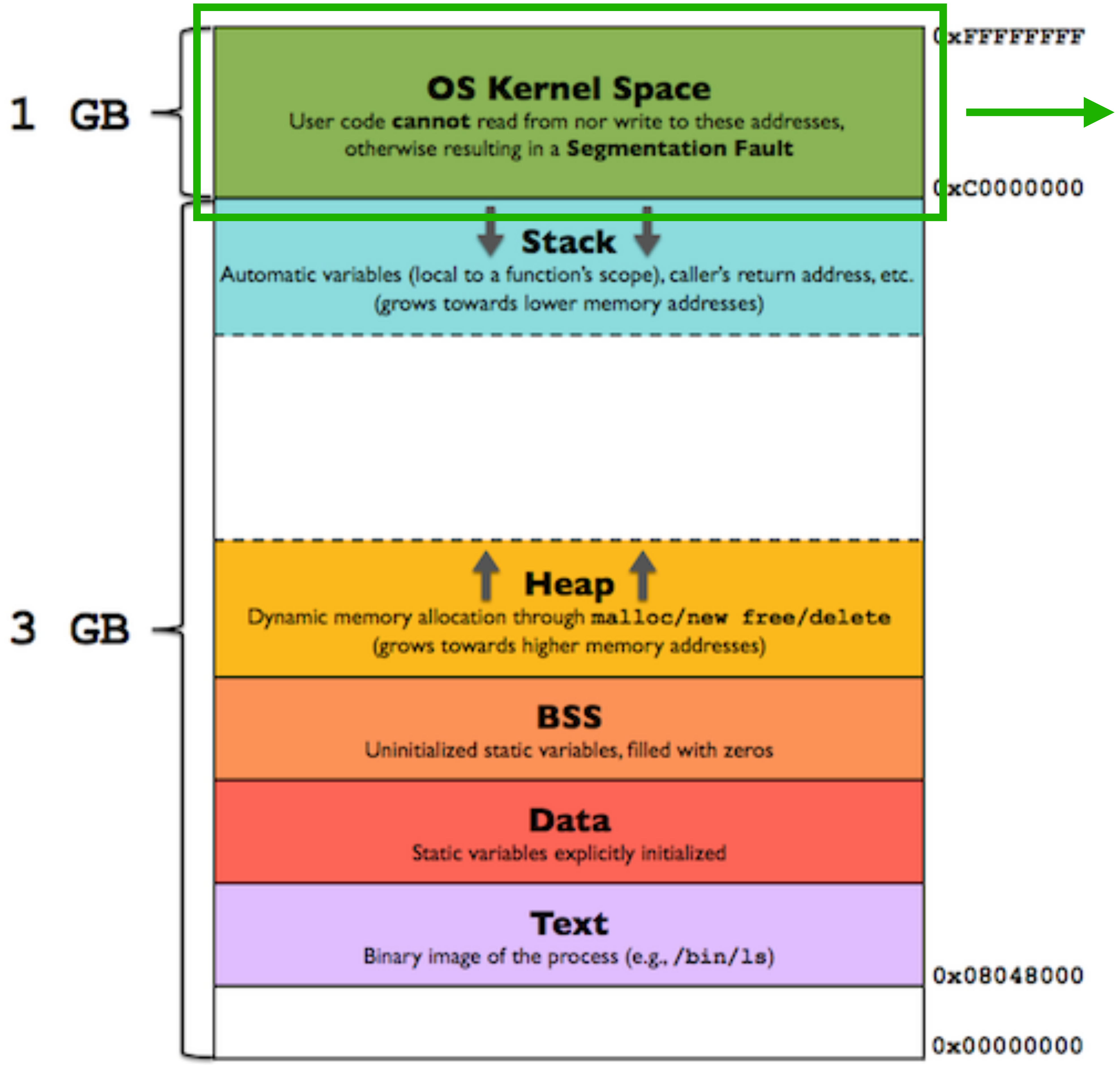
- **Kernel space** is where the core of the **operating system** (the **kernel**) runs
  - It has **full access to hardware** (e.g., CPU, memory, disks, devices)
  - Responsible for: scheduling processes, managing memory, handling I/O, enforcing security and isolation

How They Interact

User programs **cannot** just “walk” into kernel space: yhey have to ask for help through a **system call**

```
# RISC assembly pseudo-code
li   a7, 4              # Load system call code for 'print string’: in user space
la   a0, msg            # Load the address of the message: in user space
ecall                   # Trap to the OS: switch from user space to kernel space
...                     # The OS examines v0 (to know which service you’re
requesting) and a0 (the argument)

msg: .asciiz “Hello!" # Data stored in user space
```

Memory Layout 32-Bit Kernel



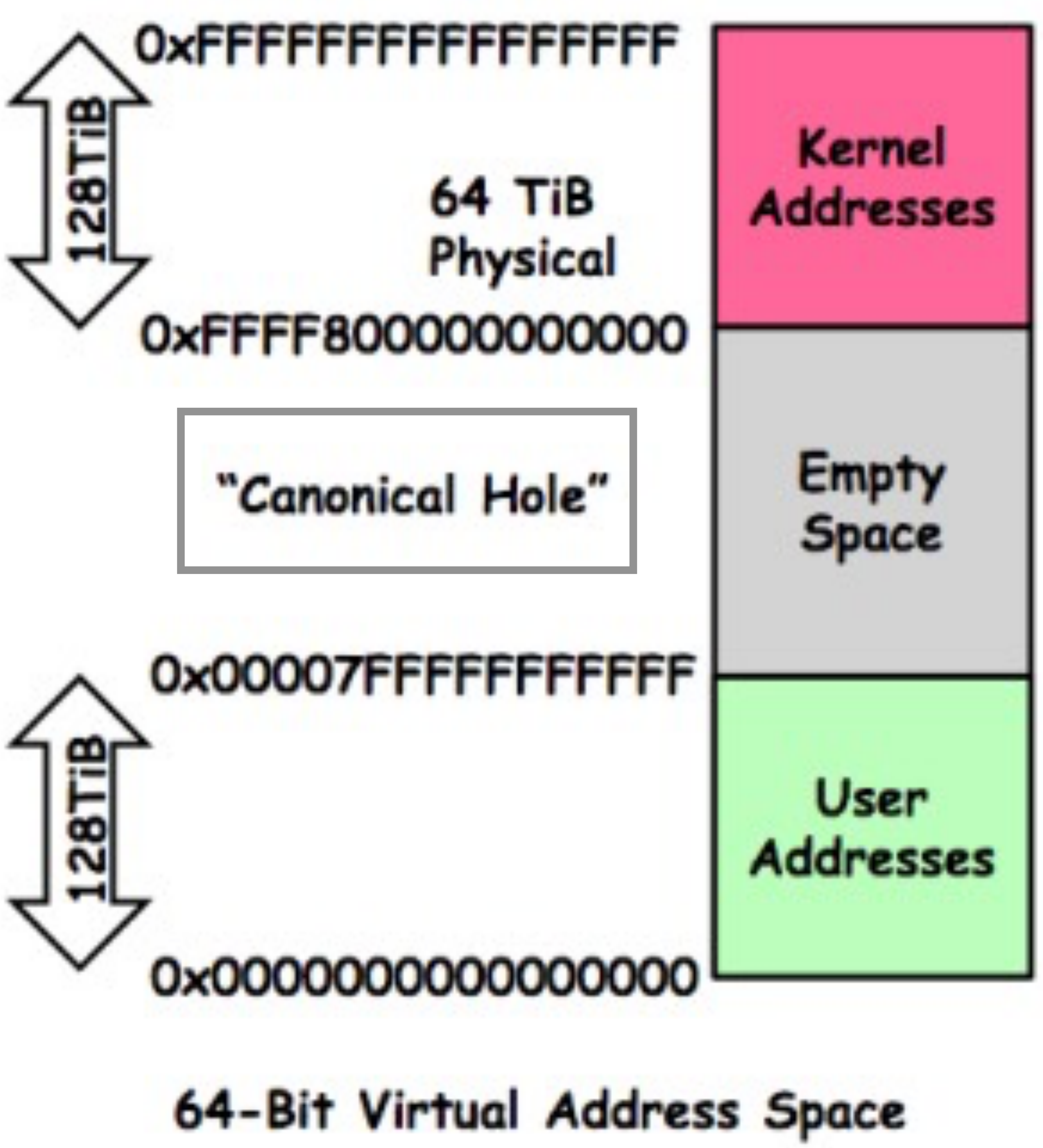
these addresses are **unavailable** in user mode

this is a **software convention**

In a 32-bit system, the **total addressable memory** is 4 GB:

- The division of memory into **1 GB for kernel space** and **3 GB for user space** is a common configuration
- It allows the OS to manage memory efficiently while providing ample space for user applications

Memory Layout 64-Bit Kernel



In a 64-bit system, the **total memory** is 16 **exabytes**:

- Current CPUs don't use all 64 bits of address lines

The available address space is **split into 2 halves separated by a very big hole** called "canonical hole"

The purpose of the **canonical hole**:

- It helps in detecting invalid memory accesses
- It enhances security and stability

Common system calls

Poll

Given a successful call to `fork()`, how many processes exist after the call, and what does the parent process receive as a return value?

[PollEv.com /gguidi](PollEv.com/gguidi)

Or send **gguidi** to **22333**

Ex: `fork()`

- `fork()` is used to create a new process by duplicating the calling process
  - The new process is called the **child** process
  - The original process is called the **parent** process

- `fork()` function prototype:

```
pid_t fork(pid_t pid);
```

- `fork()` is called, then both processes continue executing the code **after the `fork()` call**, but they have **different PIDs**

`fork()` Return Value

- `fork()` function prototype:

```
pid_t fork(pid_t pid);
```

Process	Return value of `fork()`
Parent	`PID` of the child
Child	`0`
Error	`−1`

- If `fork()` fails, it returns `−1` in the parent and no child is created

Ex: `exec()`

- `exec()` replaces the current process image with a new process image
  - Commonly used functions: `execl()`, `execp()`, `execv()`, etc.

- `exec()` function prototype:

```
int execl(const char *path, const char *arg, ...);
```

- `exec()` basically ***changes*** what a process does

Ex: exec()

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

int main() {
    printf("Before exec\n");
    execl("/bin/ls", "ls", NULL);
    perror("execl"); // this will only be executed if exec fails
    return 0;
}
```

The **`perror("execl")`** function call is used to print an error message to the standard error stream (stderr)

If **`execl()`** fails, **`perror("execl")`** will print an error message, e.g.: **`execl: No such file or directory`**

Ex: waitpid()

- `waitpid()` is used to wait for state changes in a child process
  - It can be used to wait for a specific child process to terminate

- `waitpid()` function prototype:

```
pid_t waitpid(pid_t pid, int *status, int options);
```

Return value	**Return value of `waitpid()`**
`> 0`	`PID` of the child whose state has changed
`0`	(only if `WNOHANG` used) — no child has exited yet
`-1`	-1

Ex: `waitpid()`

- `waitpid()` is used to wait for state changes in a child process
  - It can be used to wait for a specific child process to terminate

- `waitpid()` function prototype:

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- **`pid`** specifies the child process to wait for. It can take several values:
  - **`pid > 0`**: it waits for the child whose process ID is equal to **`pid`**

Ex: `waitpid()`

- `waitpid()` is used to wait for state changes in a child process
  - It can be used to wait for a specific child process to terminate

- `waitpid()` function prototype:

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- **`pid`** specifies the child process to wait for. It can take several values:
  - **`pid > 0`**: it waits for the child whose process ID is equal to **`pid`**
  - **`pid == 0`**: it waits for any child process whose **process group ID** is equal to that of the calling process

Collection of one or more processes

Ex: `waitpid()`

- `waitpid()` is used to wait for state changes in a child process
  - It can be used to wait for a specific child process to terminate

- `waitpid()` function prototype:

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- **`pid`** specifies the child process to wait for. It can take several values:
  - **`pid > 0`**: it waits for the child whose process ID is equal to **`pid`**
  - **`pid == 0`**: it waits for any child process whose **process group ID** is equal to that of the calling process
  - **`pid == -1`**: it waits for any child process; this is equivalent to **`wait()`**

Ex: `waitpid()`

- `waitpid()` is used to wait for state changes in a child process
  - It can be used to wait for a specific child process to terminate

- `waitpid()` function prototype:

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- **`pid`** specifies the child process to wait for. It can take several values:
  - **`pid > 0`**: it waits for the child whose process ID is equal to **`pid`**
  - **`pid == 0`**: it waits for any child process whose process group ID is equal to that of the calling process
  - **`pid == -1`**: it waits for any child process; this is equivalent to **`wait()`**
  - **`pid < -1`**: it waits for any child process whose process group ID is equal to the absolute value of **`pid`**

Ex: `waitpid()`

- `waitpid()` is used to wait for state changes in a child process
  - It can be used to wait for a specific child process to terminate

- `waitpid()` function prototype:

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- `status`: A pointer to an integer where the exit status of the child process will be stored.
- `options`: Provides additional options to modify the behavior of `waitpid()`

It lets the parent choose **which child** to wait for and **how** to wait.

`waitpid()` vs `wait()`

- `waitpid()` is used to wait for state changes in a child process
  - It can be used to wait for a specific child process to terminate
- `waitpid()` function prototype:

```
pid_t waitpid(pid_t pid, int *status, int options);
```

Call	It means...
`wait(&status)`	It waits for **any** child to terminate
`waitpid(-1, &status, 0)`	Same as `wait()`
`waitpid(pid, &status, 0)`	It waits for **that specific `pid` child**

Ex: waitpid()

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    pid_t pid = fork();           // create new process
    if (pid == 0) {               // if true, it means you're the child process
        printf("Child process\n"); // child prints a message to standard output
        _exit(0);                 // terminates the child process immediately with exit status 0
    } else {                      // it means you're the parent process
        int status;
        pid_t wpid = waitpid(pid, &status, 0);  // parent waits for the specific child whose PID is pid to finish
        if (wpid == -1) {         // if something went wrong, prints a system error message
            perror("waitpid");
        } else {
           if (WIFEXITED(status))
                printf("Child exited with status %d\n", WEXITSTATUS(status));
        }
    }
    return 0;
}
```

Ex: waitpid() + execlp()

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();                    // creates new process

    if (pid == 0) {
                                           // child runs "ls -l"
        execlp("ls", "ls", "-l", NULL);    // replaces the current process image with a new program
        perror("execlp failed");
    } else if (pid > 0) {
        int status;
        pid_t w = waitpid(pid, &status, 0);  // parent waits for child
        if (w == -1) {
            perror("waitpid failed");
        } else if (WIFEXITED(status)) {
            printf("Child exited with status %d\n", WEXITSTATUS(status));
    }
}
```

How Processes Are Created?

- `fork()`:
  - It allocates the process ID **`pid`**
  - Create and initialize PCB
  - Create and initialize a new address space
  - Then, inform the scheduler a new process is ready to run

How Processes Are Created?

- `fork()`:
  - It allocates the process ID **`pid`**
  - Create and initialize PCB
  - Create and initialize a new address space
  - Then, inform the scheduler a new process is ready to run
- `exec(program, arguments)`:
  - Load the program into the address space
  - Copy arguments into memory address space
  - Then, initialize hardware context to stat execution at “start”

How Processes Are Terminated?

- The system calls for termination are:
  - **`exit()`**: used by a process to terminate itself
  - **`abort()`**: used by a parent process to terminate a child process
  - **`wait()`** and **`waitpid()`**: used by a parent process to wait for the termination of a child process and retrieve its exit status

False Friend `kill()`

- Despite its name, `kill()` doesn't necessarily *kill* a process; it can send **any** signal, including ones that don't terminate the target

- `kill()` function prototype:

```
int kill(pid_t pid, int sig);
```

Outcome	**Return value of `kill()`**
Success	`0`
Error	`-1`

fork() + exec()

Process 1
Program A

PC

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

child_pid ?

Child process doesn’t exist yet

fork() + exec()

Process 1
Program A

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

child_pid 42 → Child PID in parent process

Process 42
Program A

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

child_pid 0 → Child PID in child process

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 A

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

PC

child_pid 0

Fork returns twice!

fork() + exec()

Process 1
Program A

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

PC

Waits until the child process exits

child_pid 42

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
  wait(&status);
```

PC

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 excuted!*

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

```
main() {
   ...

   exit(3);
}
```

PC

fork() + exec()

Process 1
Program A

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

PC

child_pid 42

status 3

Brief Summary

- A **process** is an abstraction of a computer
- A process is **not** a program
- A **context** captures the state of the processor
- The **implementation** uses two spaces: user space and kernel space
- A **Process Control Block (PCB)** is a kernel data structure that saves context and has other information about the process

System Signals

A **signal** is away for processes to communicate with each other

Common signals:

A signal sent to the processor by hardware or software indicating an event that needs immediate attention

- **SIGINT**: the **interrupt** signal
- **SIGTERM**: the termination signal is used to request a process to terminate
- **SIGKILL**: the kill signal forces a process to terminate immediately; **cannot be caught or ignored**
- **SIGSEGV**: the segmentation fault (e.g., process tries to access an invalid memory location)
- **SIGCHLD**: it's sent to a parent process when a child process terminates
- **SIGALRM**: it's alarm clock signal (e.g., timer)

Sending Signals

You can send signals using the `kill()` function or the `raise()` function:

```
kill(pid_t pid, int sig)
```

`kill()` sends the signal **sig** to the process with the process ID **pid**

```
raise(int sig)
```

`raise()` sends the signal **sig** to the calling process itself

Interrupt

An **interrupt** is a signal sent to the processor by hardware or software indicating an event that needs immediate attention

Types of Interrupt

- **Hardware Interrupt:** Generated by external hardware devices to get the CPU's attention
    - Keyboard input (key press)
    - Mouse movement or click
    - Disk I/O completion
    - Network packet arrival

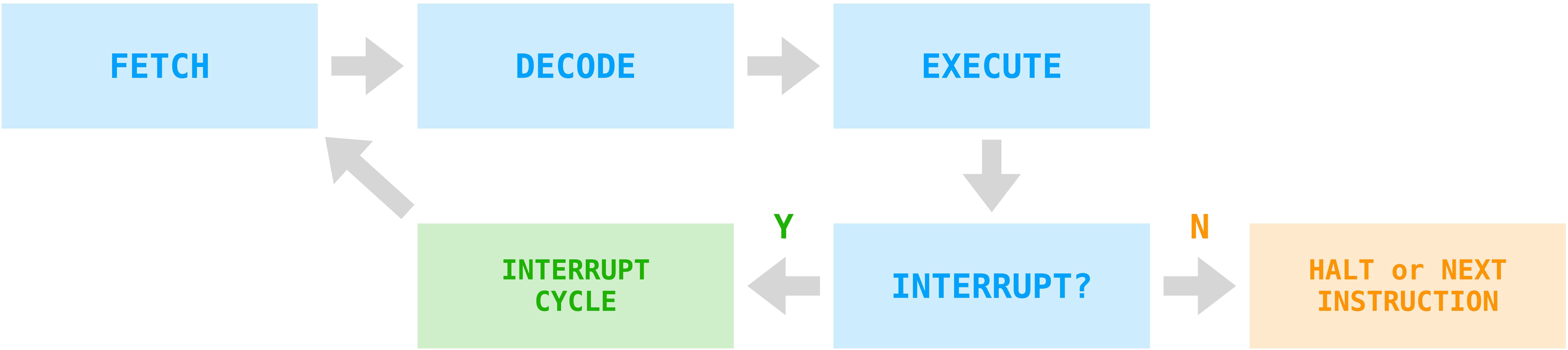
Type	Description
Maskable Interrupt	Can be temporarily "masked" (disabled) by the CPU; used for regular hardware events
Non-Maskable Interrupt	Cannot be disabled; reserved for critical events like hardware failure or power issues

Types of Interrupt

- **Software Interrupt**: Generated by a program or the operating system, typically to request a system service or perform an exception
    - System calls (e.g., syscall instruction)
    - Exceptions like divide-by-zero or invalid memory access
    - Breakpoints during debugging

Type	Description
System Call Interrupt	Triggered by a program to request OS services (e.g., file read/write)
Exception Interrupt	Generated automatically when an error or specific condition occurs (divide by zero, page fault)
Software-Generated Signal	Explicitly triggered in software (e.g., raise(), kill() in Unix)

Instruction Cycle and Interrupt



How Interrupts Work

- **Interrupt Signal**
    - An interrupt signal is sent to the CPU by a hardware device or software

How Interrupts Work

- **Interrupt Signal**
    - An interrupt signal is sent to the CPU by a hardware device or software
- **Saving State**
    - The CPU saves the current state of the running process (e.g., program counter, registers) so it can resume execution later

How Interrupts Work

- **Interrupt Signal**
  - An interrupt signal is sent to the CPU by a hardware device or software
- **Saving State**
  - The CPU saves the current state of the running process (e.g., program counter, registers) so it can resume execution later
- **Interrupt Handling**
  - The CPU transfers control to the interrupt handler associated with the interrupt. The interrupt handler processes the event (e.g., reading data from a device)

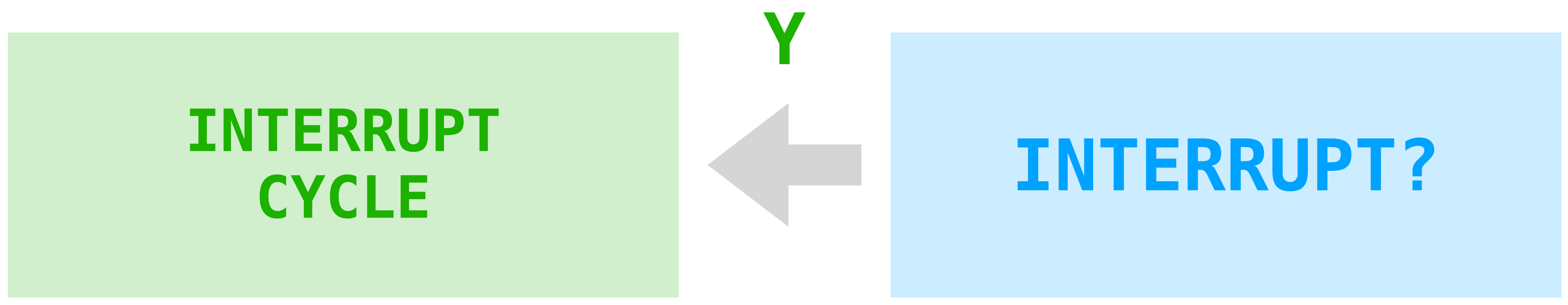
How Interrupts Work

- **Interrupt Signal**
  - An interrupt signal is sent to the CPU by a hardware device or software
- **Saving State**
  - The CPU saves the current state of the running process (e.g., program counter, registers) so it can resume execution later
- **Interrupt Handling**
  - The CPU transfers control to the interrupt handler associated with the interrupt. The interrupt handler processes the event (e.g., reading data from a device)
- **Restoring State**
  - The CPU restores the saved state and resumes execution of the interrupted process

Why Interrupt?

- **Efficiency,** → To avoid constant polling; CPU works while waiting for I/O
- **Responsiveness,** → React immediately to I/O and user input
- And, **multitasking** → Timer interrupts let OS switch between processes

How does a system call actually work?



How System Calls Work

They act as the **bridge** between user-level applications and the kernel:

- **System Call Request**
  - The application requests a system call by invoking a corresponding function provided by the operating system's API
- **Switch to Kernel Mode**
  - The CPU switches from user mode to kernel mode
- **Execution in Kernel**
  - The kernel receives the system call request and performs the necessary operations
- **Return to User Mode**
  - The kernel switches the CPU back to user mode and returns control to the calling process

Executing a System Call

- **Process**
  1. Calls systems 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. Then, executes `syscall` or `ecall` (RISC-V) machine instruction
- **Kernel**
  1. Executes `syscall` or `ecall` (RISC-V) interrupt handler
  2. Places result in a dedicated register
  3. Executes `return_from_interrupt` interrupt handler
- **Process**
  1. Executes `return_from_function`

Conceptual RISC-V Print "Hello"

Conventionally, **a7 holds the system call number in RISC-V**: it tells the OS **which** service the program is asking for

```
# RISC assembly pseudo-code
li   a7, 4              # Load system call code for 'print string'
la   a0, msg            # Load address of message
ecall                   # Call to the OS, equivalent to syscall
...
msg: .asciiz "Hello!"
```

PC moves from **user** space to **kernel** space (more on this later)

This is a **special instruction** that triggers a **trap** to the operating system (OS)

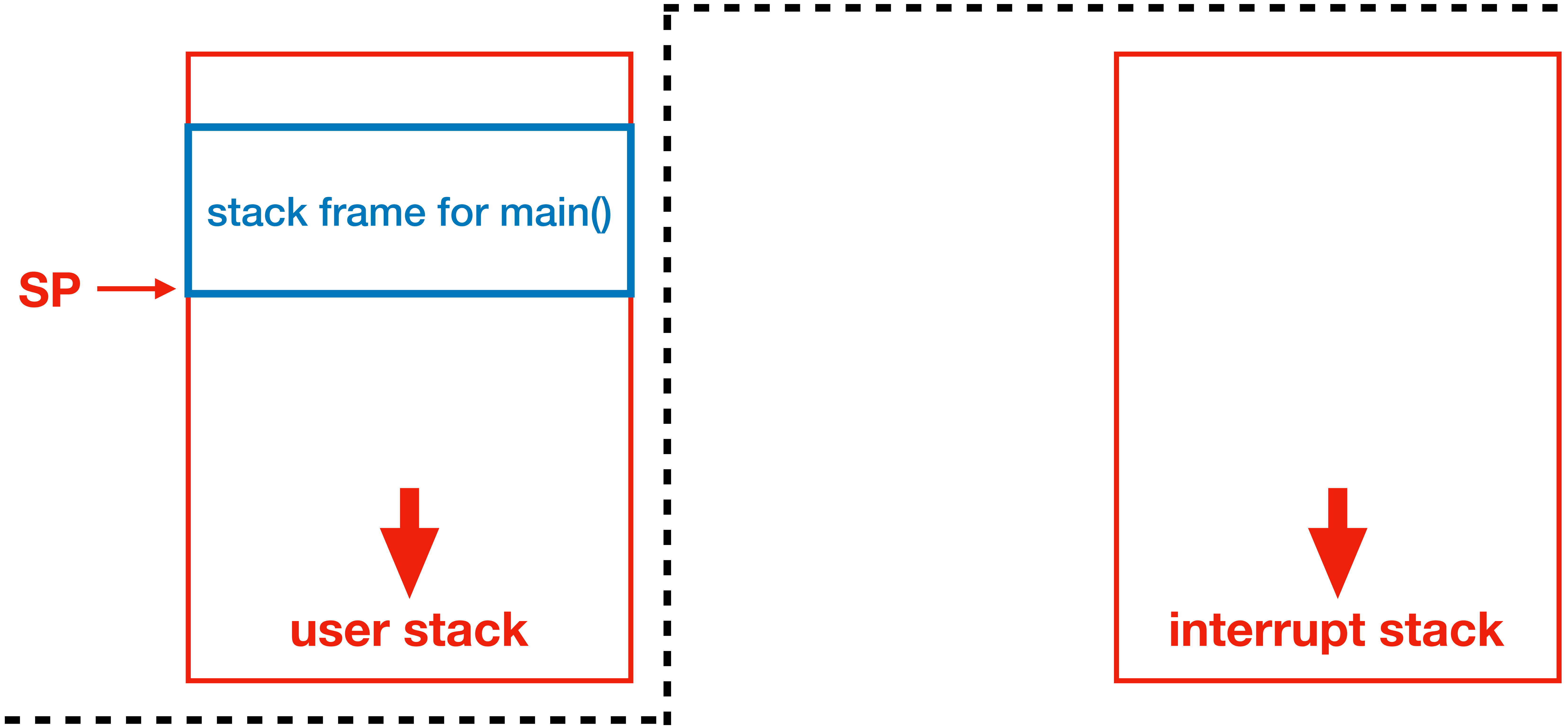
- The OS looks at **a7** (system call number = 4) and `a0` (address of string)
- It performs the requested service: writing "`Hello!`" to the terminal

Executing `read` System Call

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

Executing `read` System Call

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

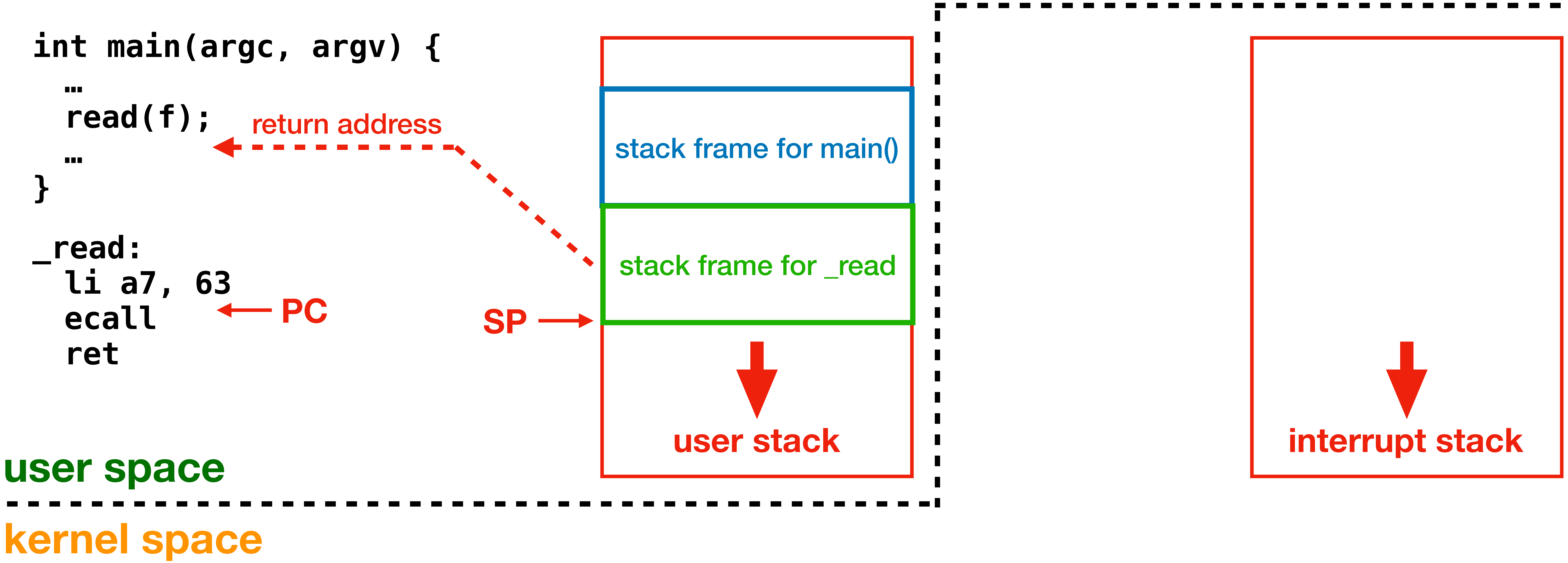


user space

kernel space

Executing `read` System Call

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

_read:
  li a7, 63
  ecall
  ret
```



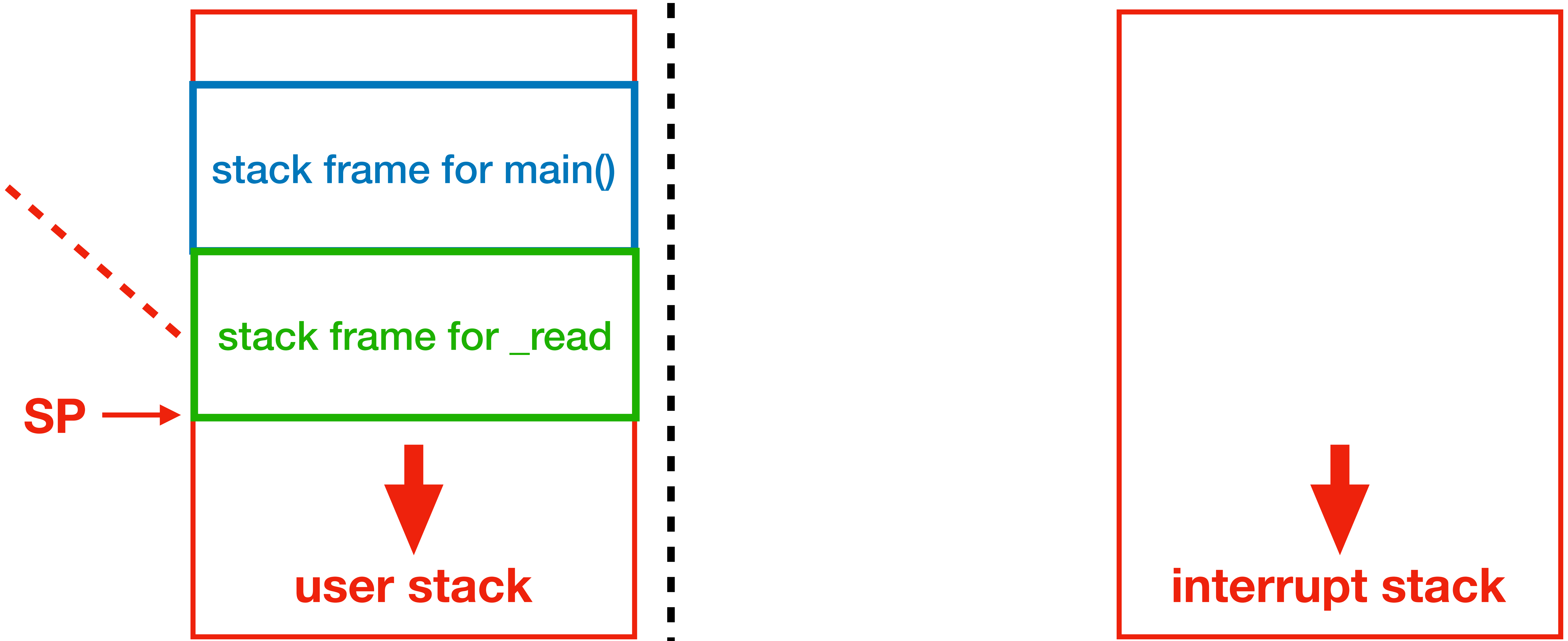
In RISC-V, the equivalent of the **`syscall`** instruction found in some other architectures is called **`ecall`** (Environment Call)

Executing `read` System Call

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

return address

```
_read:
  li a7, 63
  ecall
  ret
```



user space

kernel space

It saved the PC

PCB: sp

```
HandleIntrSyscall:
  mv %sp, $pcb
  mv $stacktop, %sp
  push %Rn
  …
  call _HandleSyscall
  …
  pop %Rn
  mv  $pcb, &sp
  return_from_interrupt
```

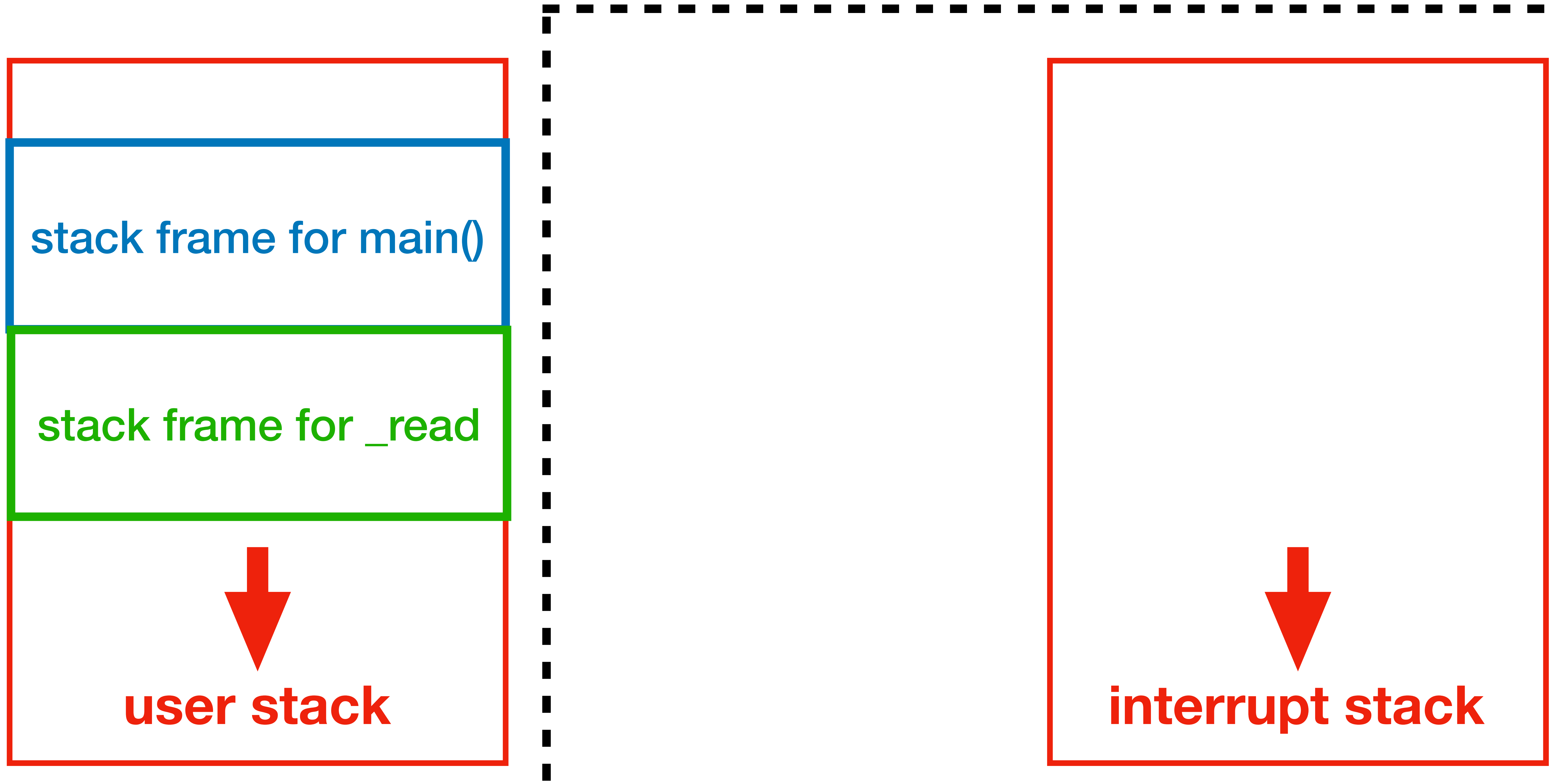
PC

Executing `read` System Call

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

_read:
  li a7, 63
  ecall
  ret
```

return address

SP



user space

kernel space

It saved the PC

PCB: sp

```
HandleIntrSyscall:
   mv %sp, $pcb
   mv $stacktop, %sp
   push %Rn
   …
   call _HandleSyscall
   …
   pop %Rn
   mv  $pcb, &sp
   return_from_interrupt
```

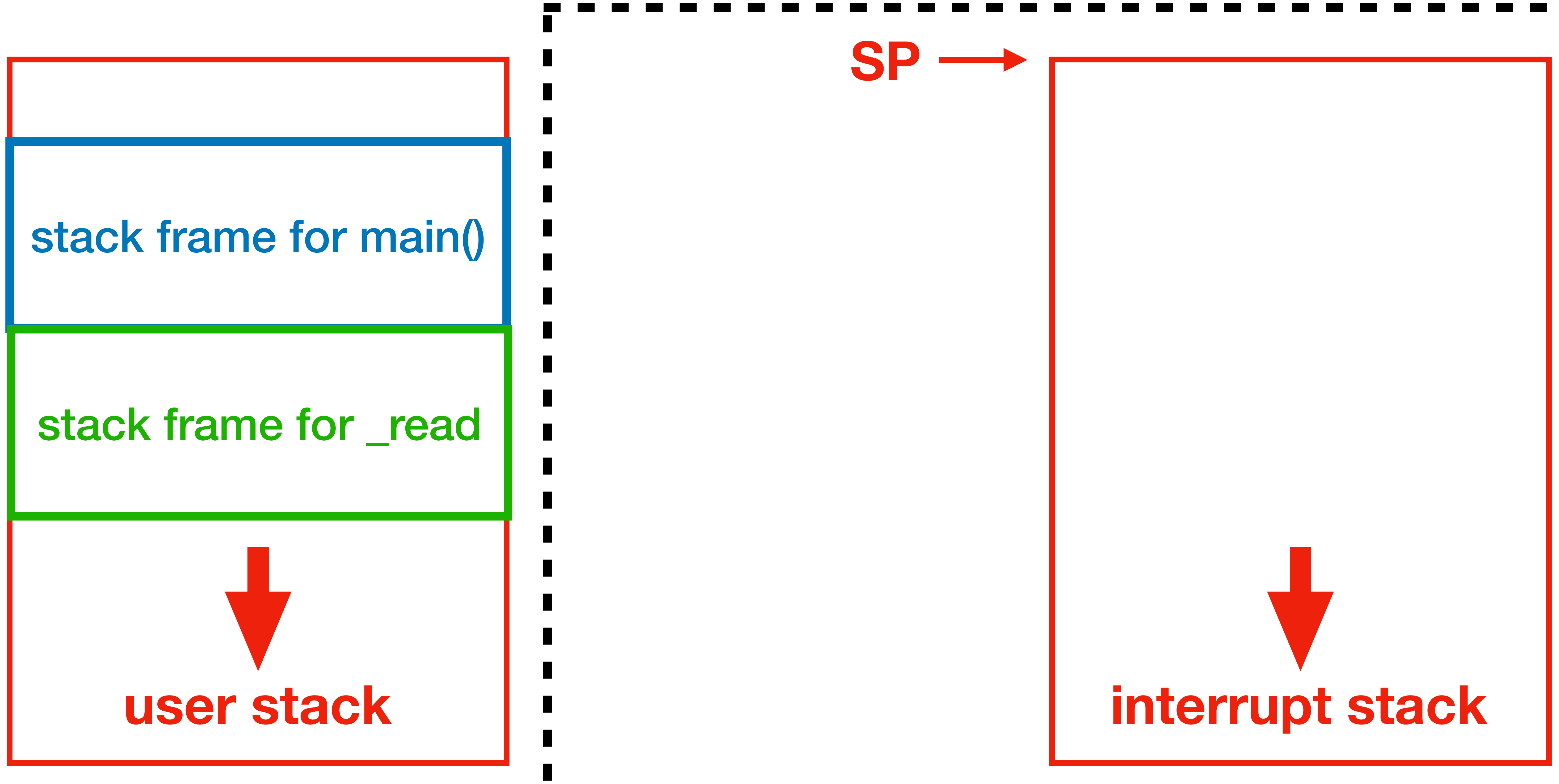
PC

Executing `read` System Call

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

return address

```
_read:
  li a7, 63
  ecall
  ret
```



user space

kernel space

It saved the PC

PCB: sp

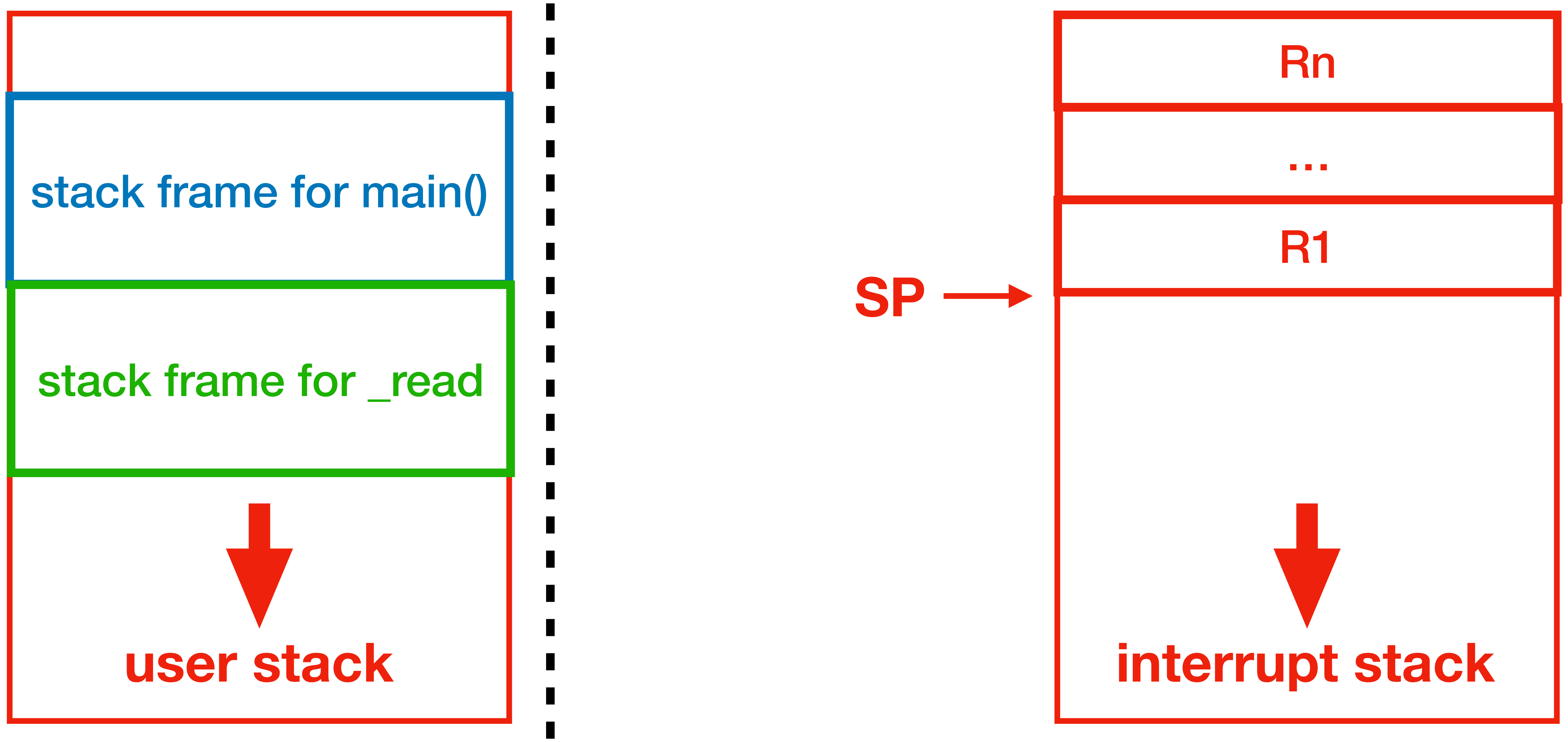
```
HandleIntrSyscall:
   mv %sp, $pcb
   mv $stacktop, %sp
   push %Rn
   …
   call _HandleSyscall
   …
   pop %Rn
   mv  $pcb, &sp
   return_from_interrupt
```

PC

Executing `read` System Call

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

_read:
  li a7, 63
  ecall
  ret
```

return address



user space

kernel space

It saved the PC

PCB: sp

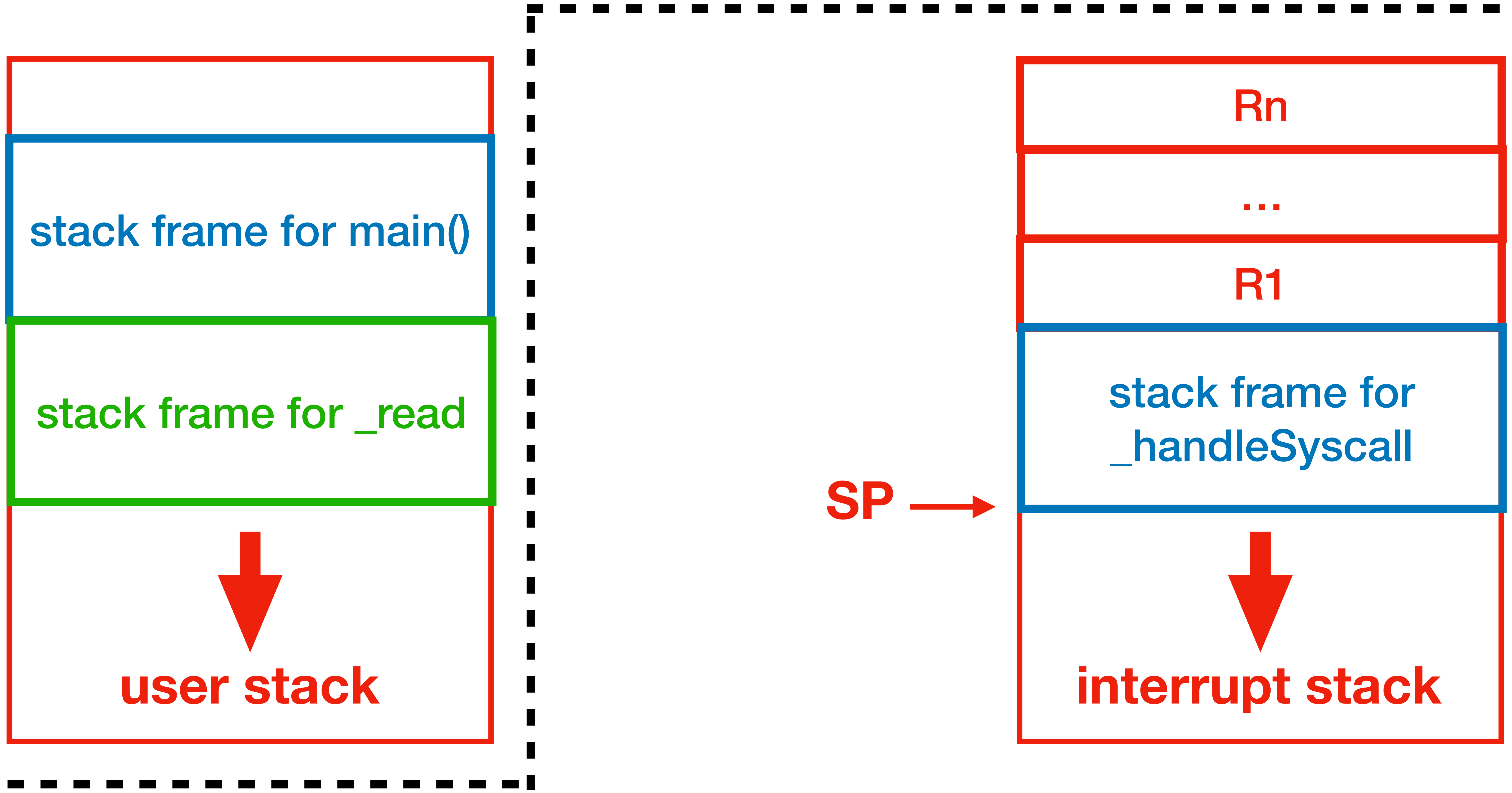
```
HandleIntrSyscall:
   mv %sp, $pcb
   mv $stacktop, %sp
   push %Rn
   …
   call _HandleSyscall
   …
   pop %Rn
   mv  $pcb, &sp
   return_from_interrupt
```

PC

Executing `read` System Call

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

_read:
  li a7, 63
  ecall
  ret
```

return address



user space

kernel space

It saved the PC

PCB: sp

```
HandleIntrSyscall:
   mv %sp, $pcb
   mv $stacktop, %sp
   push %Rn
   …
   call _handleSyscall
   …
   pop %Rn
   mv  $pcb, &sp
   return_from_interrupt
```

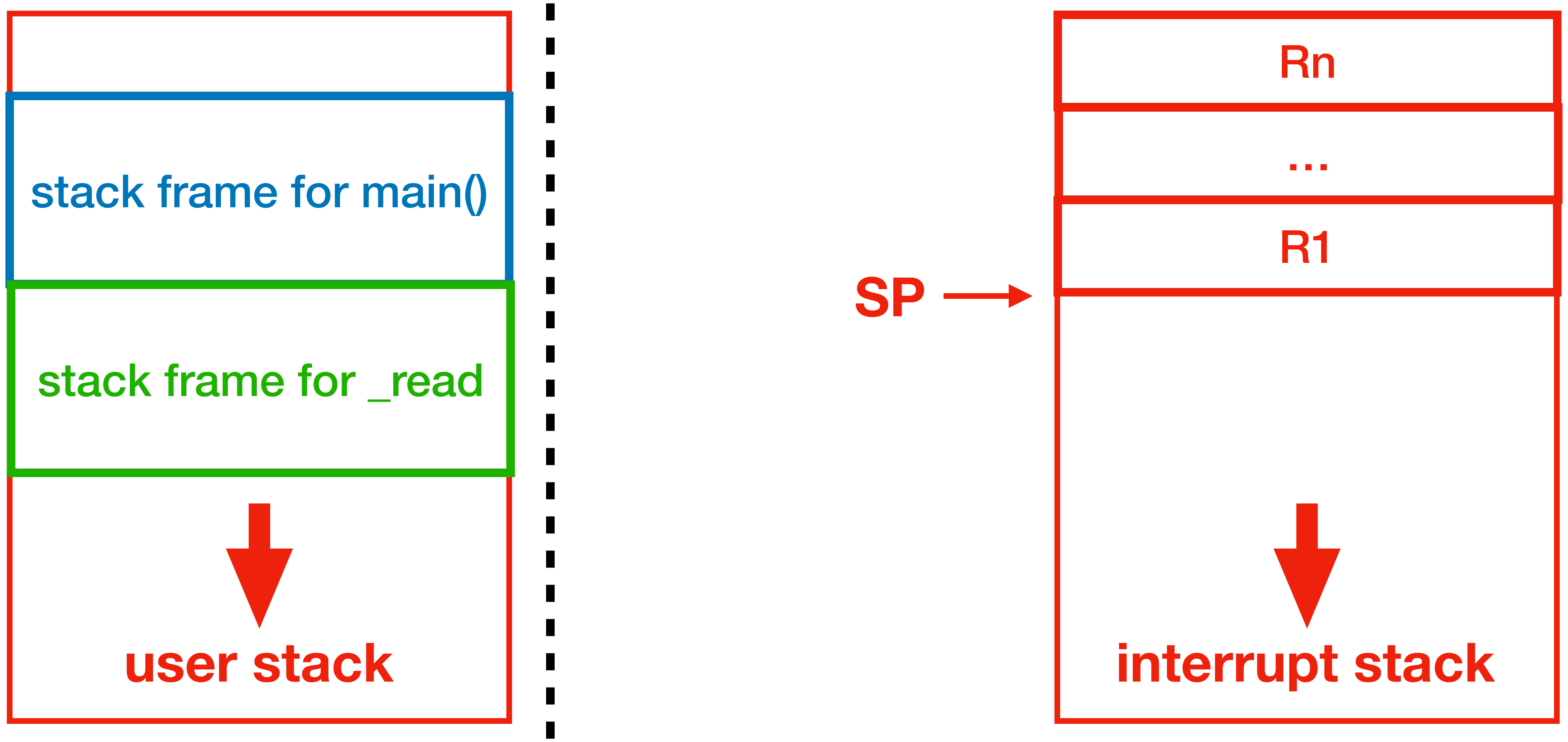
```
int handleSyscall(int type) {
    switch (type)
    {
      case READ: …
    }
}
```

PC

Executing `read` System Call

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

_read:
  li a7, 63
  ecall
  ret
```

return address



user space

kernel space

It saved the PC

PCB: sp

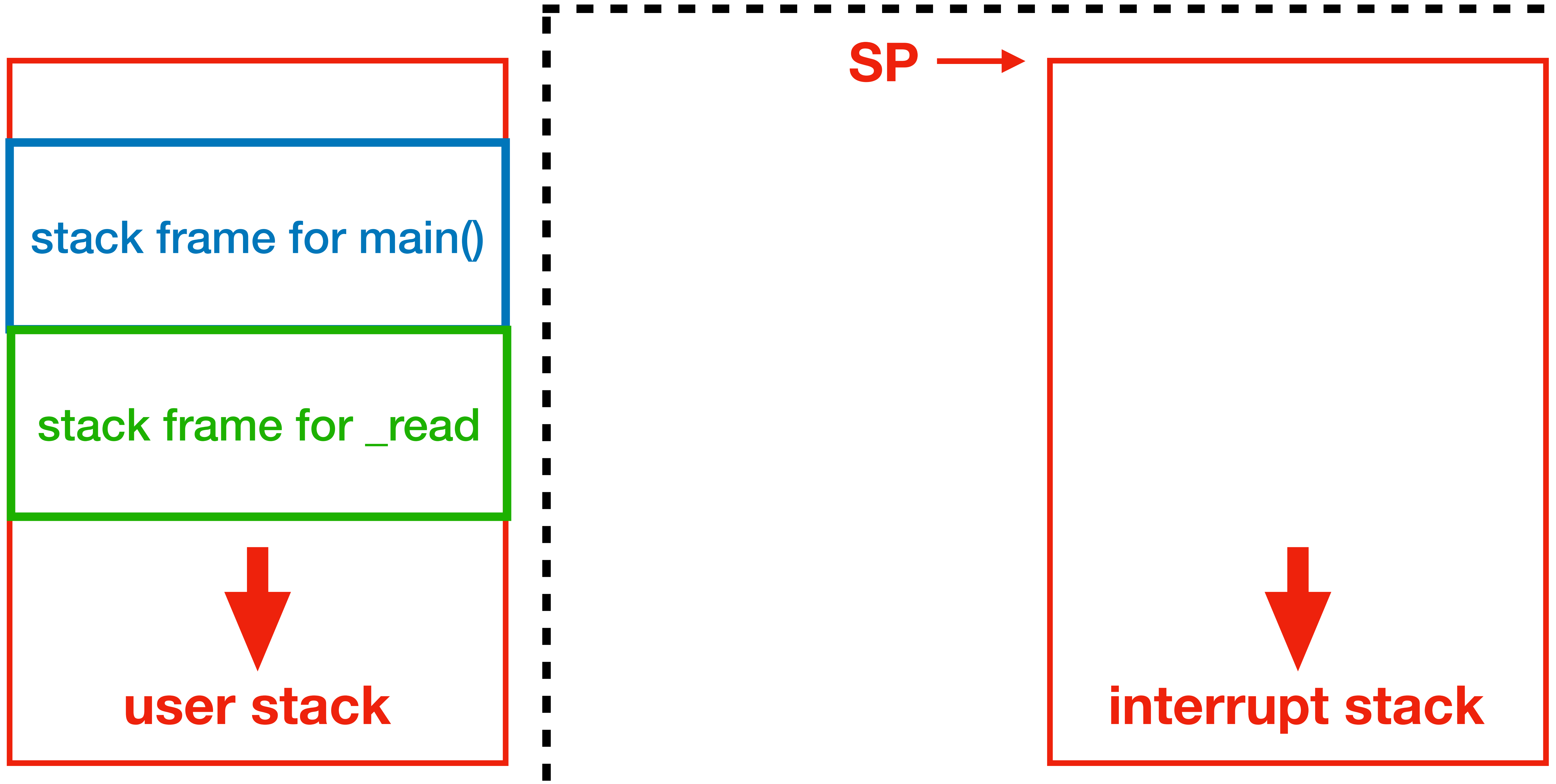
```
HandleIntrSyscall:
   mv %sp, $pcb
   mv $stacktop, %sp
   push %Rn
   …
   call _handleSyscall
   …
   pop %Rn
   mv  $pcb, &sp
   return_from_interrupt
```

PC

Executing `read` System Call

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

_read:
  li a7, 63
  ecall
  ret
```

return address



user space

kernel space

It saved the PC

PCB: sp

```
HandleIntrSyscall:
  mv %sp, $pcb
  mv $stacktop, %sp
  push %Rn
  …
  call _handleSyscall
  …
  pop %Rn
  mv  $pcb, &sp
  return_from_interrupt
```

PC

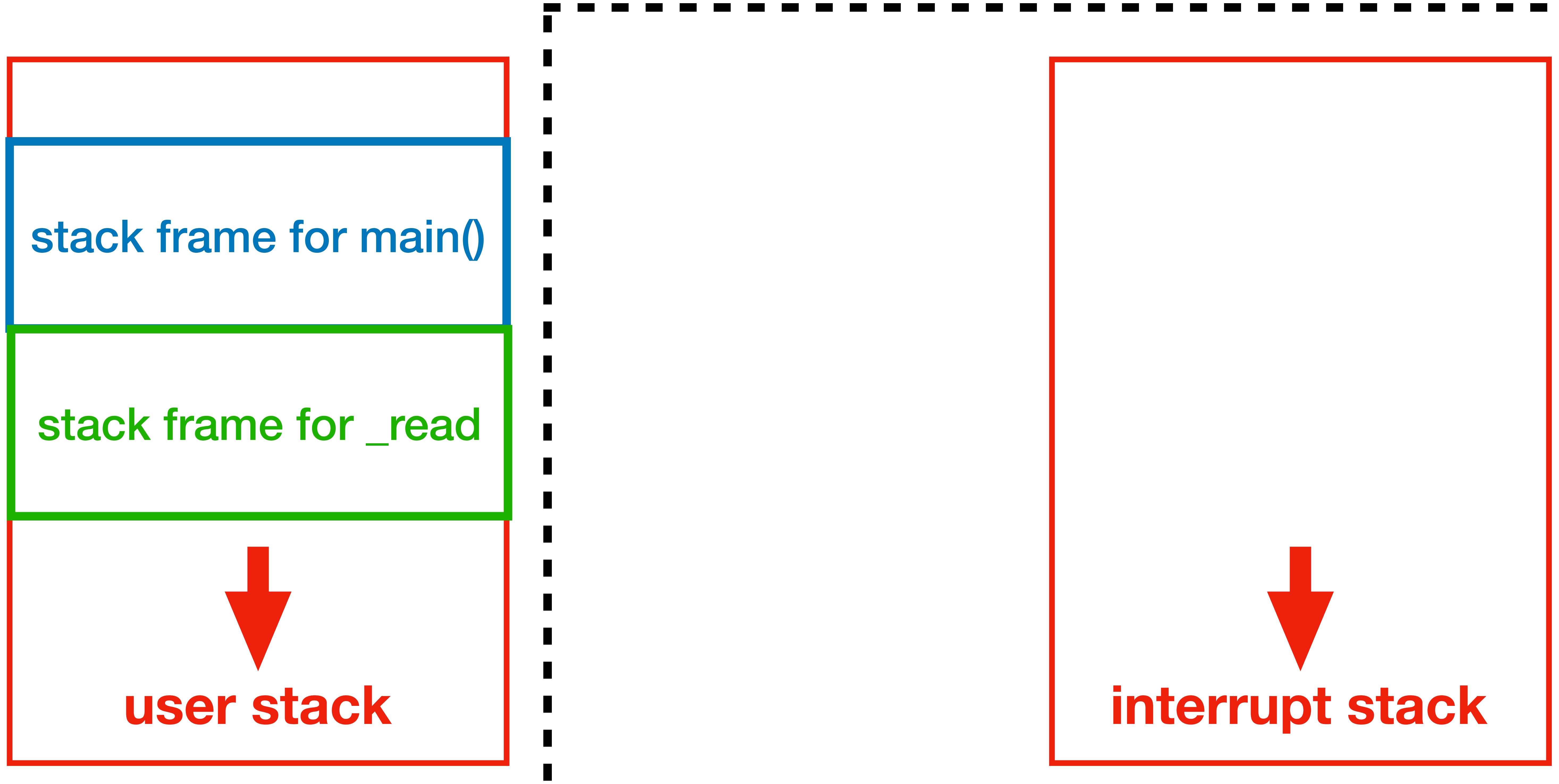
Executing `read` System Call

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

return address

```
_read:
  li a7, 63
  ecall
  ret
```

SP



user space

kernel space

It saved the PC

PCB: sp

```
HandleIntrSyscall:
   mv %sp, $pcb
   mv $stacktop, %sp
   push %Rn
   …
   call _HandleSyscall
   …
   pop %Rn
   mv  $pcb, &sp
   return_from_interrupt
```

PC

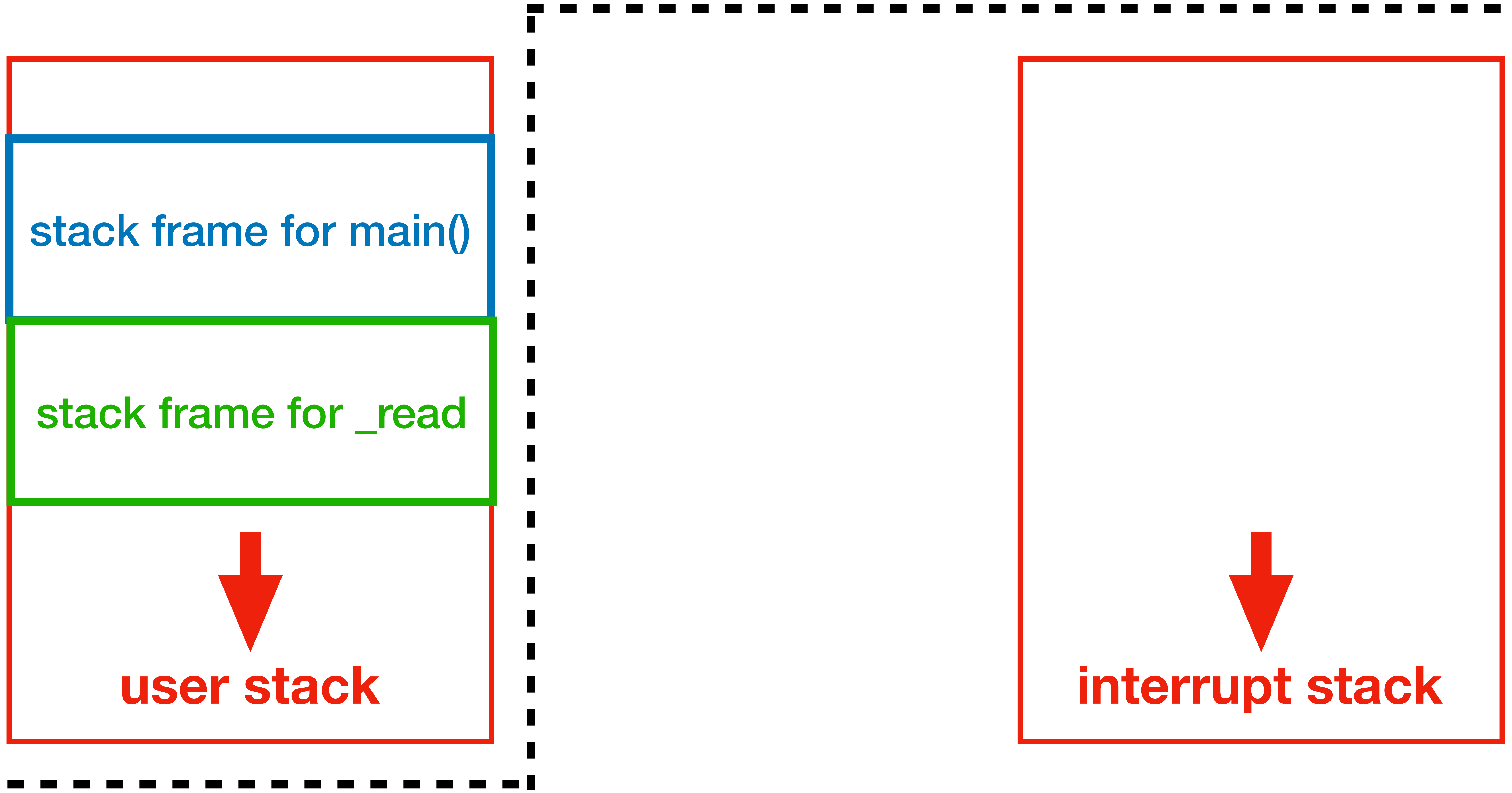
Executing `read` System Call

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

_read:
  li a7, 63
  ecall
  ret
```

return address

PC

SP



user space

kernel space

It saved the PC

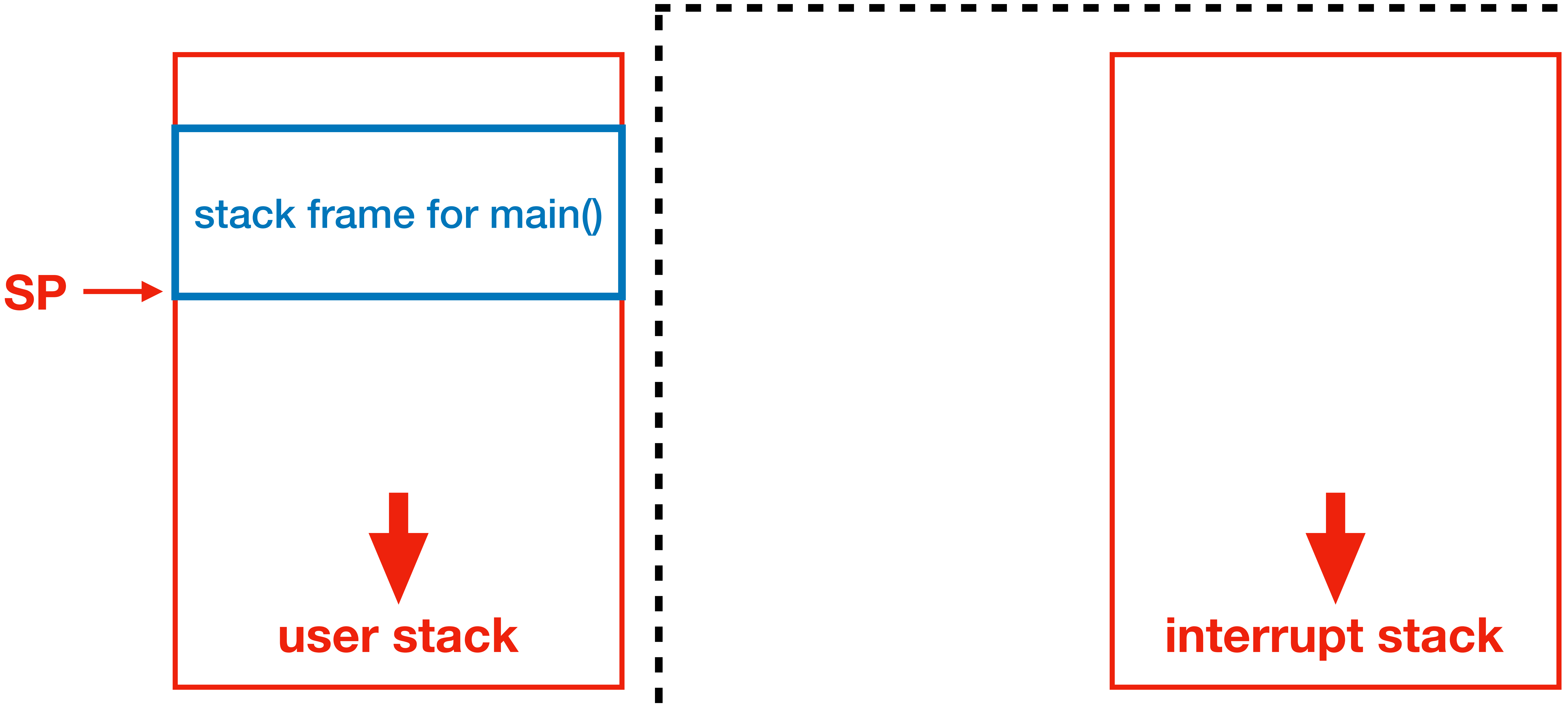
PCB: sp

```
HandleIntrSyscall:
  mv %sp, $pcb
  mv $stacktop, %sp
  push %Rn
  …
  call _HandleSyscall
  …
  pop %Rn
  mv  $pcb, &sp
  return_from_interrupt
```

Executing `read` System Call

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

_read:
  li a7, 63
  ecall
  ret
```

PC



user space

kernel space

The Invariants to Remember

- Per core, at most 1 process is RUNNING at any time
- If 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) or on some wait queue (if WAITING)
- If process is FINISHED
    - Its PCB is on finished queue