CS 423
Operating System Design
https://cs423-uiuc.github.io

Tianyin Xu
tyxu@illinois.edu

* Thanks Adam Bates and Ram Alagappan for the slides.

Overview

Process concept
- A process is the OS abstraction for executing a program with limited privileges

Dual-mode operation: user vs. kernel
- Kernel-mode: execute with complete privileges
- User-mode: execute with fewer privileges

Safe control transfer
- How do we switch from one mode to the other?

Process Abstraction

Process: an instance of a program that runs with limited rights on the machine

- Thread: a sequence of instructions within a process
  - Potentially many threads per process (for now, assume 1:1)
- Address space: set of rights of a process
  - Memory that the process can access
  - Other permissions the process has (e.g., which system calls it can make, what files it can access)

Thought Experiment

How can we permit a process to execute with only limited privileges?

Thought Experiment

How can we implement execution with limited privilege?

- Execute each program instruction in a simulator
- If the instruction is permitted, do the instruction
- Otherwise, stop the process
- Basic model in Javascript and other interpreted languages

Thought Experiment

How can we implement execution with limited privilege?

- Execute each program instruction in a simulator
- If the instruction is permitted, do the instruction
- Otherwise, stop the process
- Basic model in Javascript and other interpreted languages

Ok… but how do we go faster?

Thought Experiment

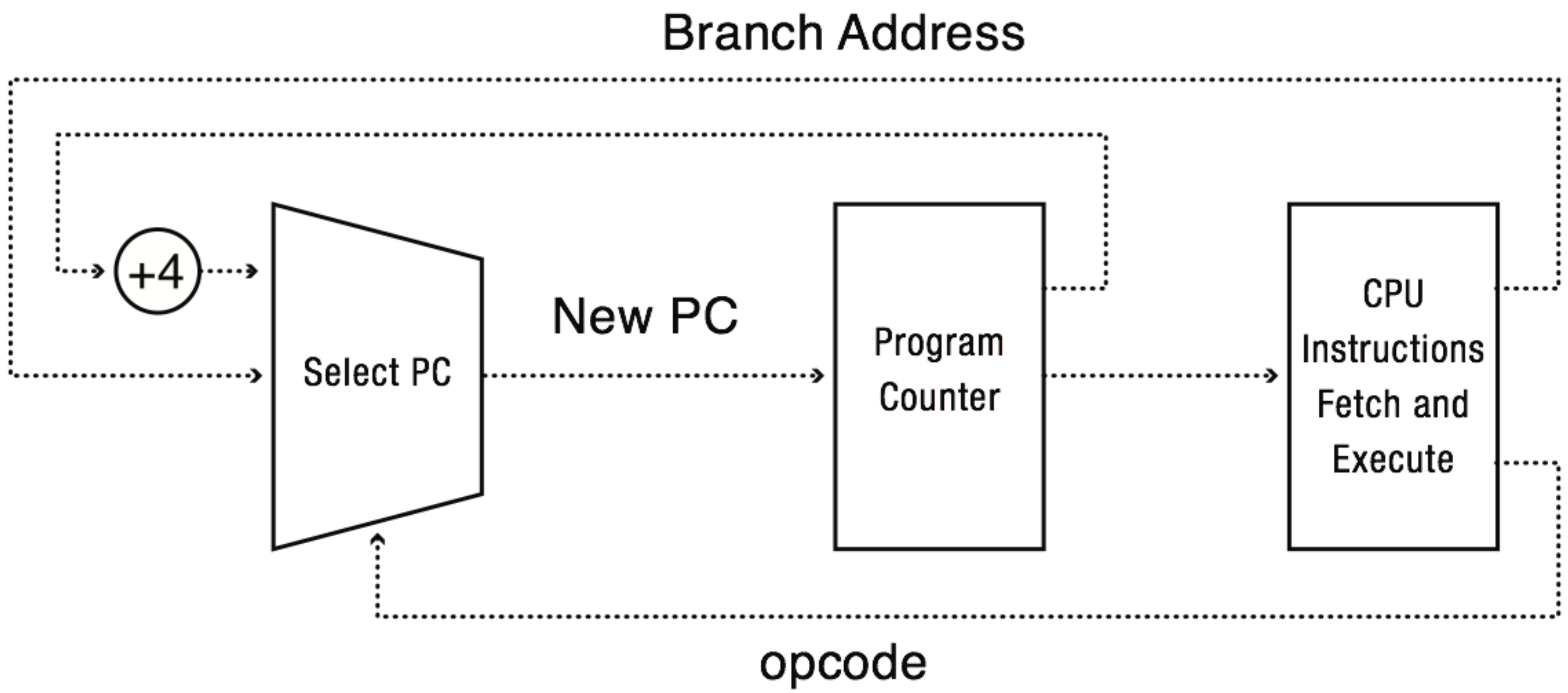
How can we implement execution with limited privilege?

- Execute each program instruction in a simulator
- If the instruction is permitted, do the instruction
- Otherwise, stop the process
- Basic model in Javascript and other interpreted languages

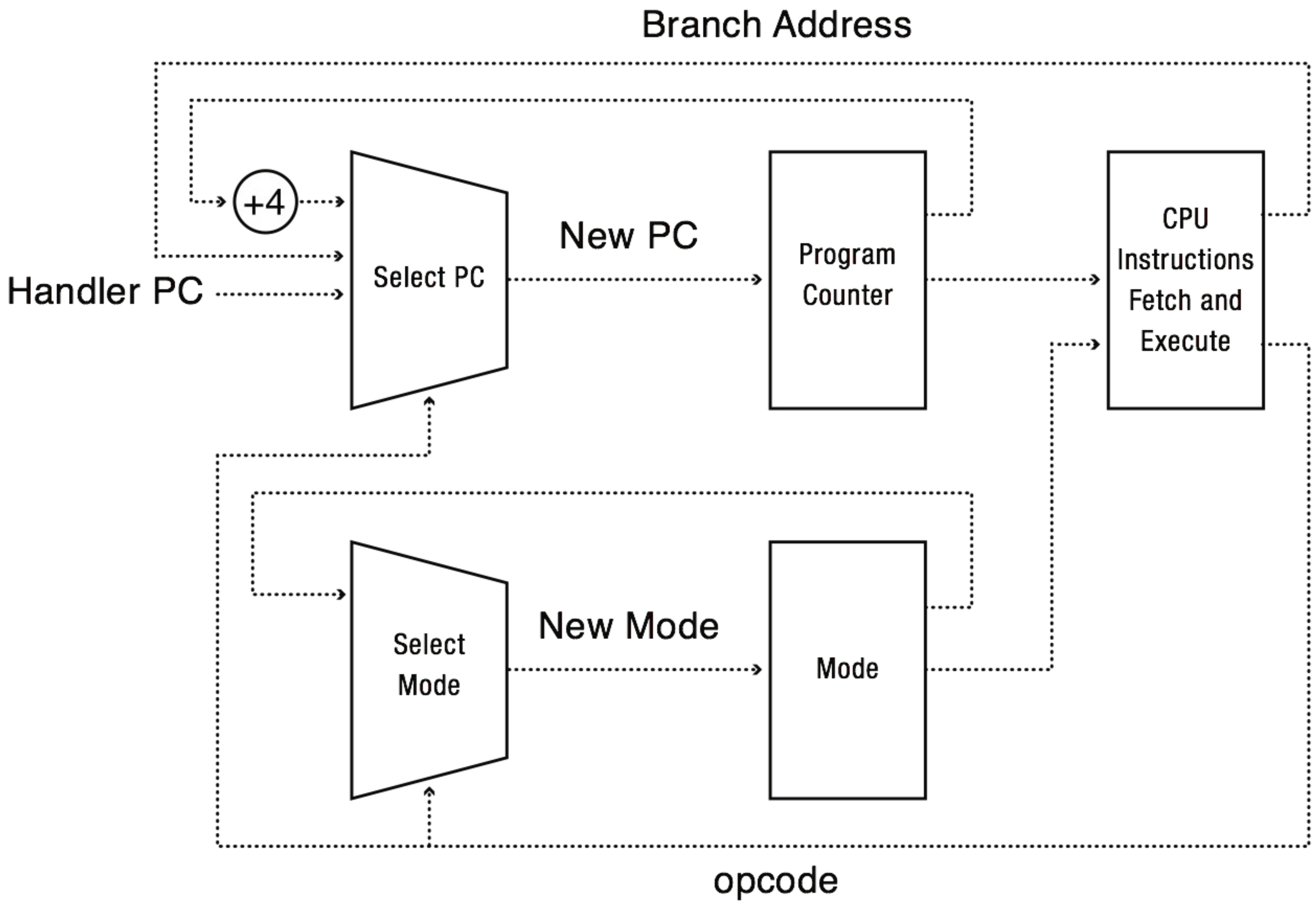
Ok… but how do we go faster?

- Run the unprivileged code directly on the CPU!

A Model of a CPU



A CPU with Dual-Mode Operation



HW Support for Dual-Mode

Privileged instructions
- Available to kernel
- Not available to user code

Limits on memory accesses
- To prevent user code from overwriting the kernel

Timer
- To regain control from a user program in a loop

Safe way to switch from user mode to kernel mode, and vice versa

Privileged Instructions

Examples?

What should happen if a user program attempts to execute a privileged instruction?

User->Kernel Switches

How/when do we switch from user to kernel mode?

1. Interrupts
   - Triggered by timer and I/O devices
2. Exceptions
   - Triggered by unexpected program behavior
   - Or malicious behavior!
3. System calls (aka protected procedure call)
   - Request by program for kernel to do some operation on its behalf
   - Only limited # of very carefully coded entry points

Question

How does the OS know when a process is in an infinite loop?

Hardware Timer

Hardware device that periodically interrupts the processor

- Returns control to the kernel handler
- Interrupt frequency set by the kernel
  Not by user code!
- Interrupts can be temporarily deferred
  Not by user code! Interrupt deferral crucial for implementing mutual exclusion

Kernel->User Switches

How/when do we switch from kernel to user mode?

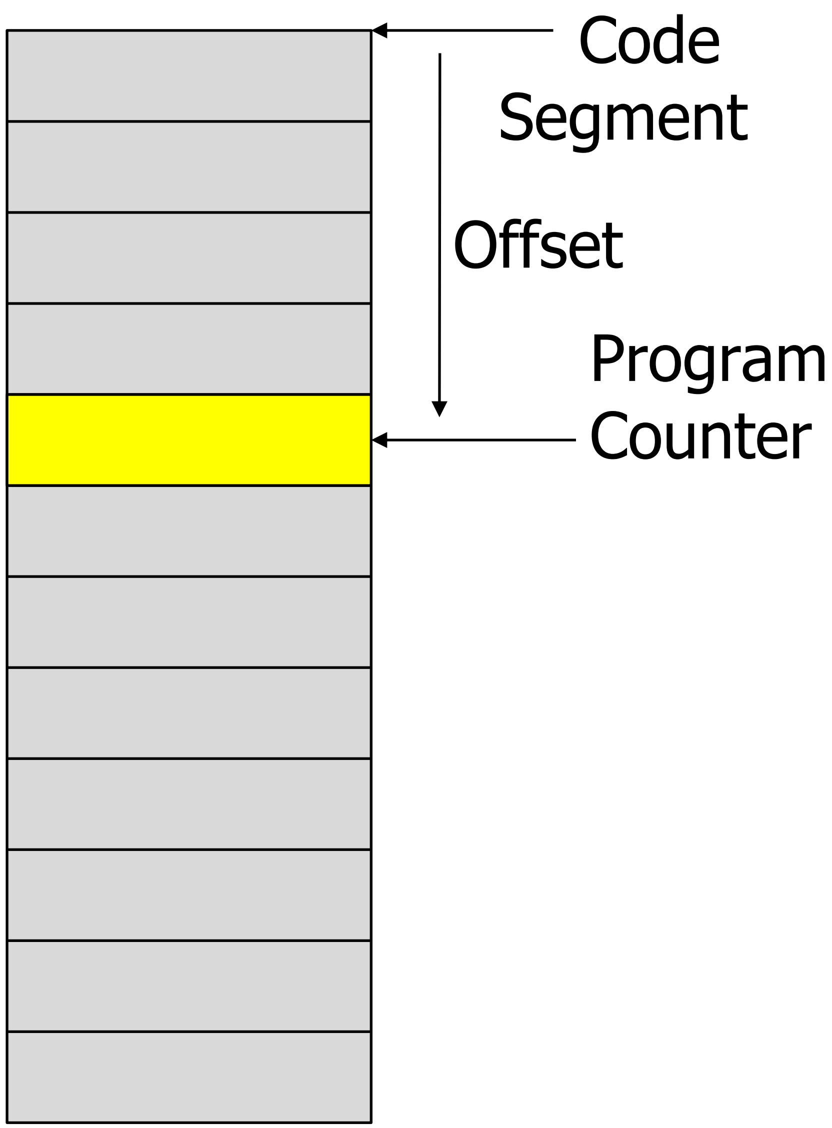
1. New process/new thread start
   - Jump to first instruction in program/thread
2. Return from interrupt, exception, system call
   - Resume suspended execution (return to PC)
3. Process/thread context switch
   - Resume some other process (return to PC)
4. User-level upcall (UNIX signal)
   - Asynchronous notification to user program

CPU State

What is the CPU's behavior defined by at any given moment?

CPU State

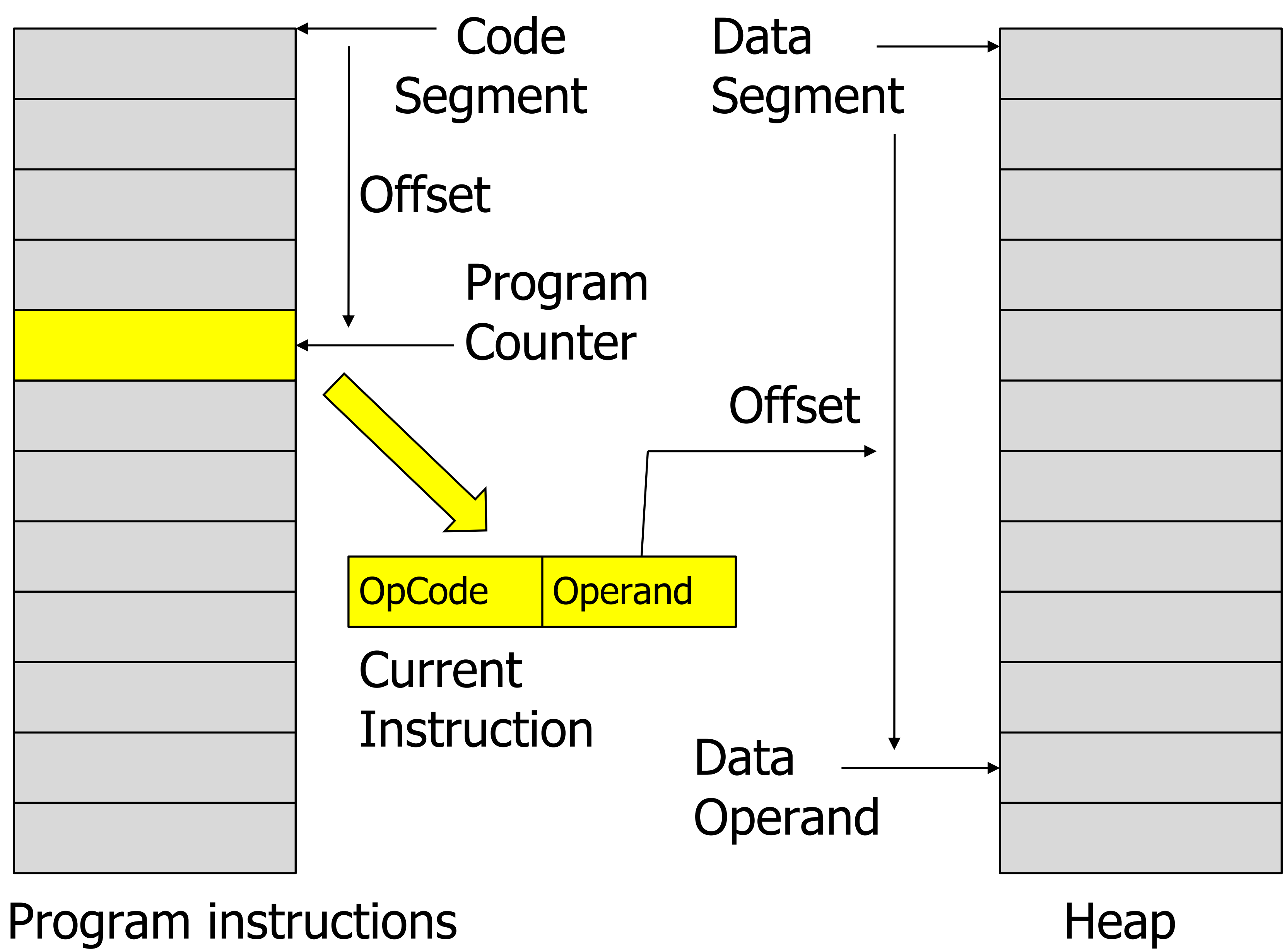
What is the CPU's behavior defined by at any given moment?



Program instructions

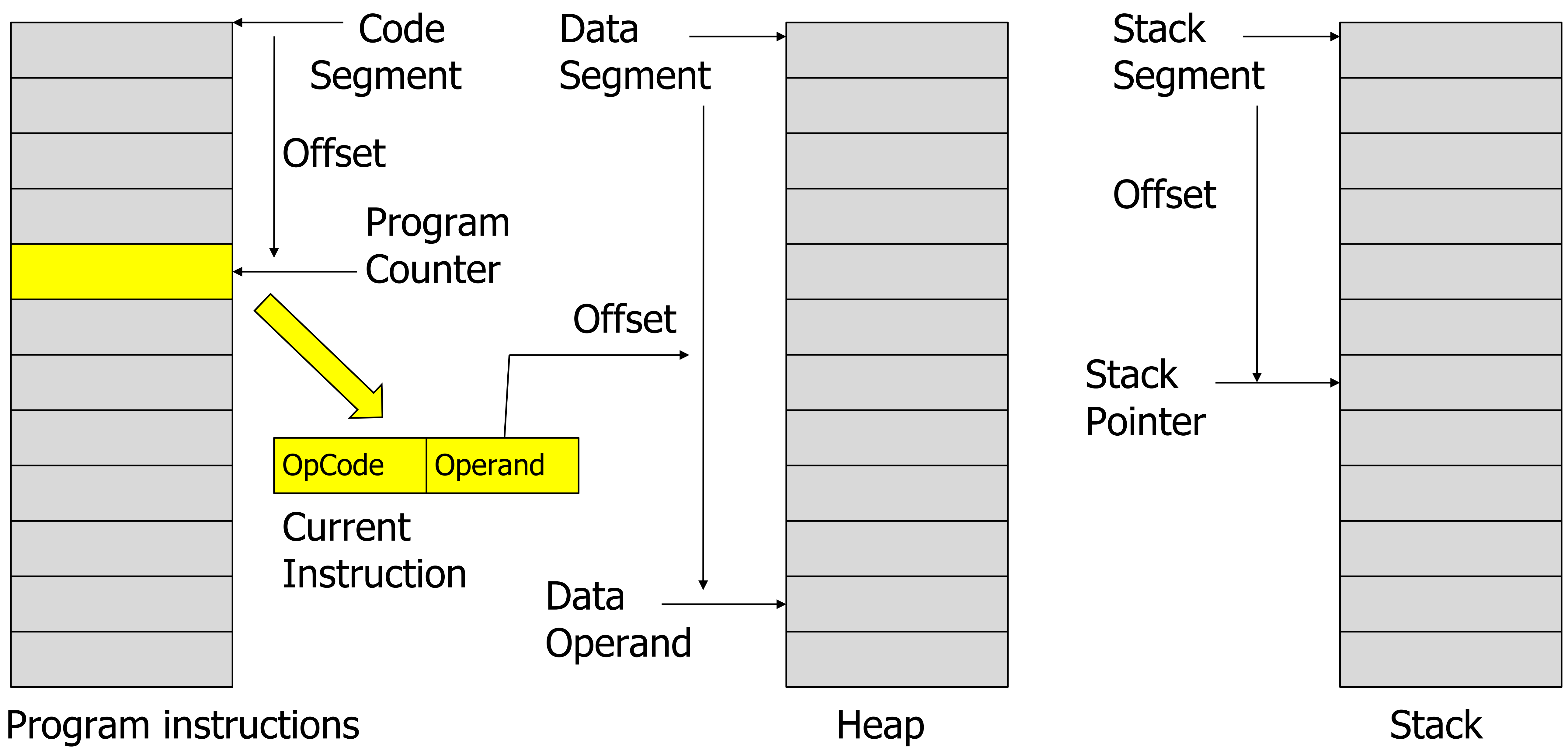
CPU State

What is the CPU's behavior defined by at any given moment?



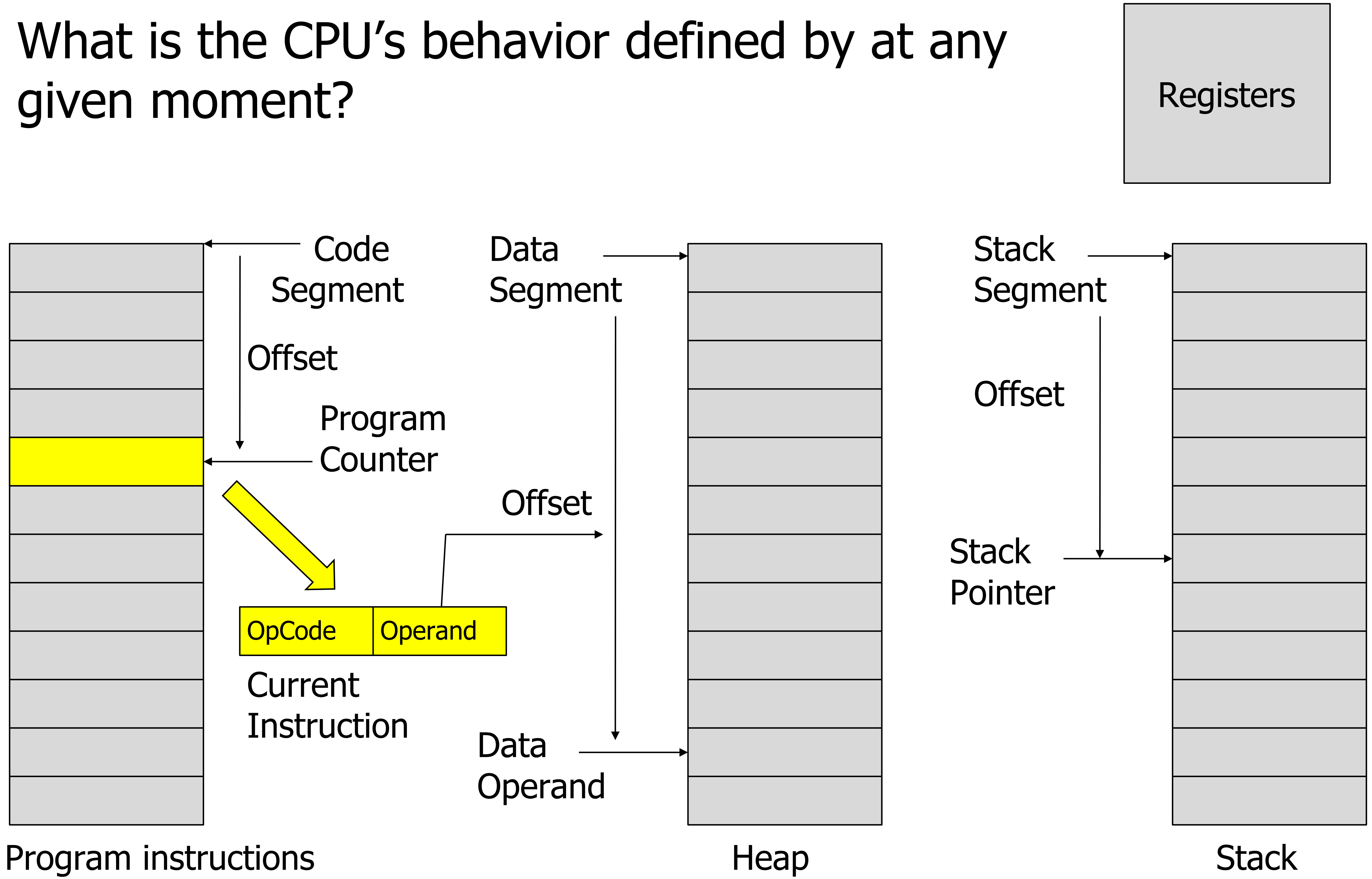
CPU State

What is the CPU's behavior defined by at any given moment?



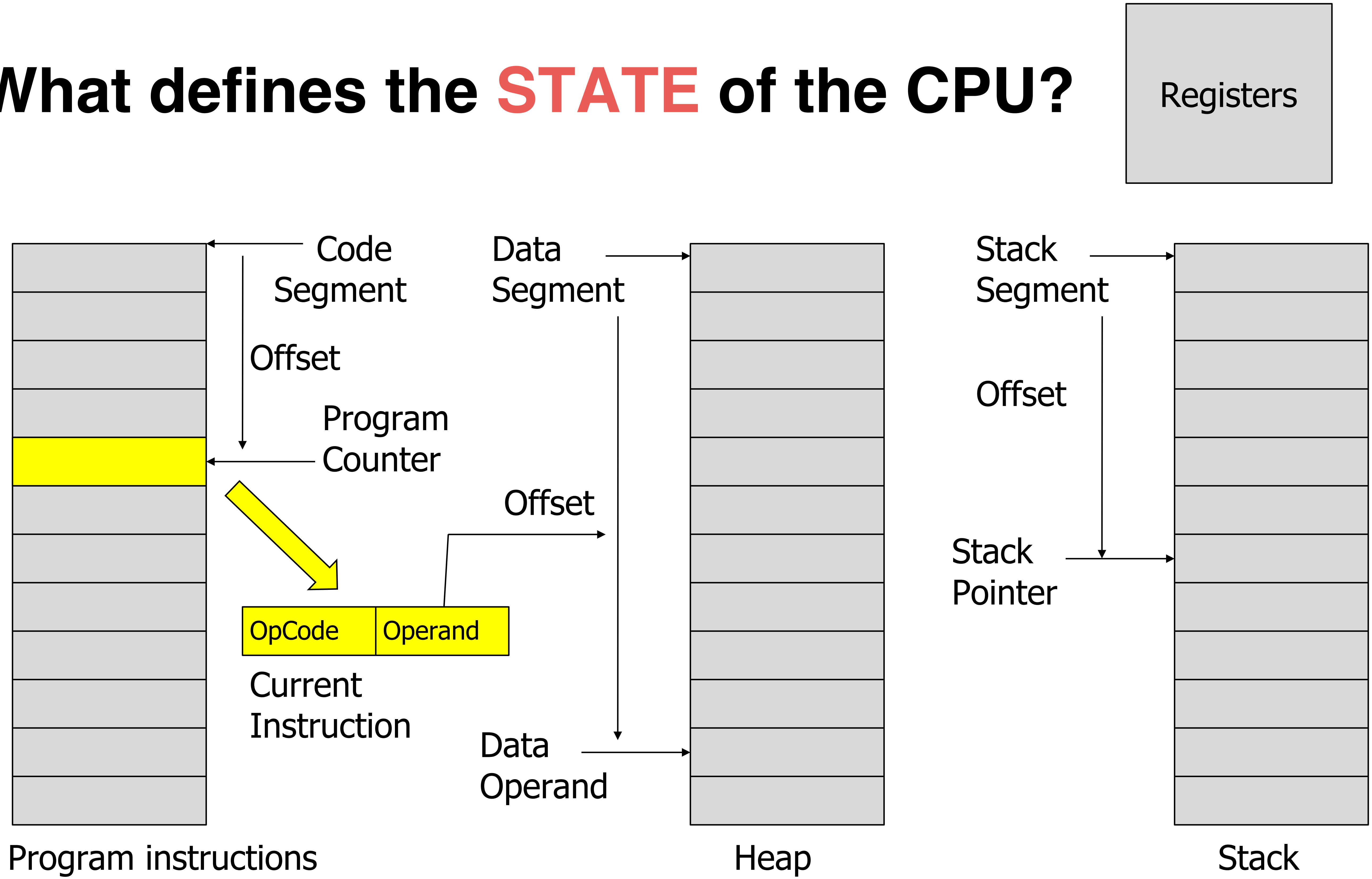
CPU State

What is the CPU's behavior defined by at any given moment?



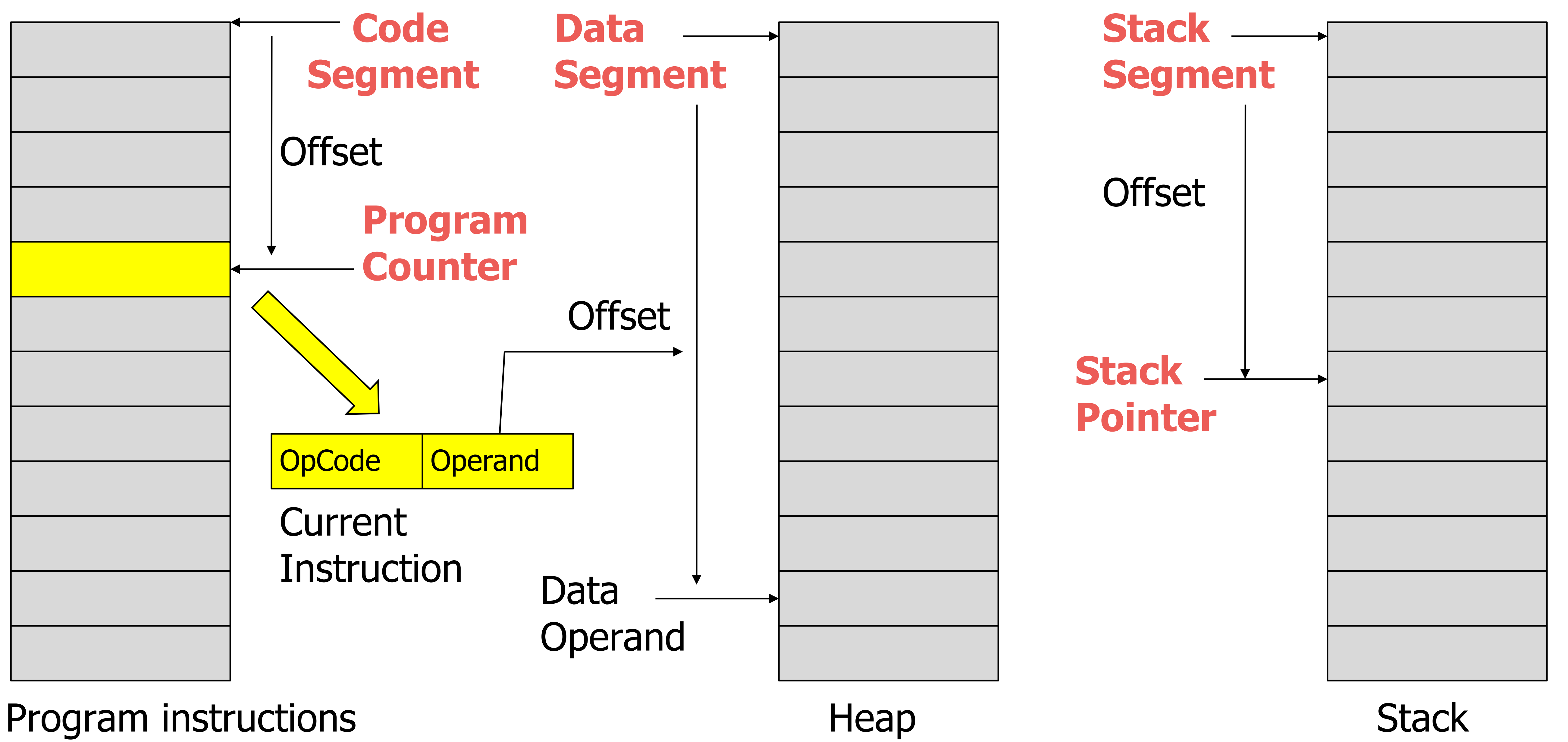
CPU State

What defines the STATE of the CPU?

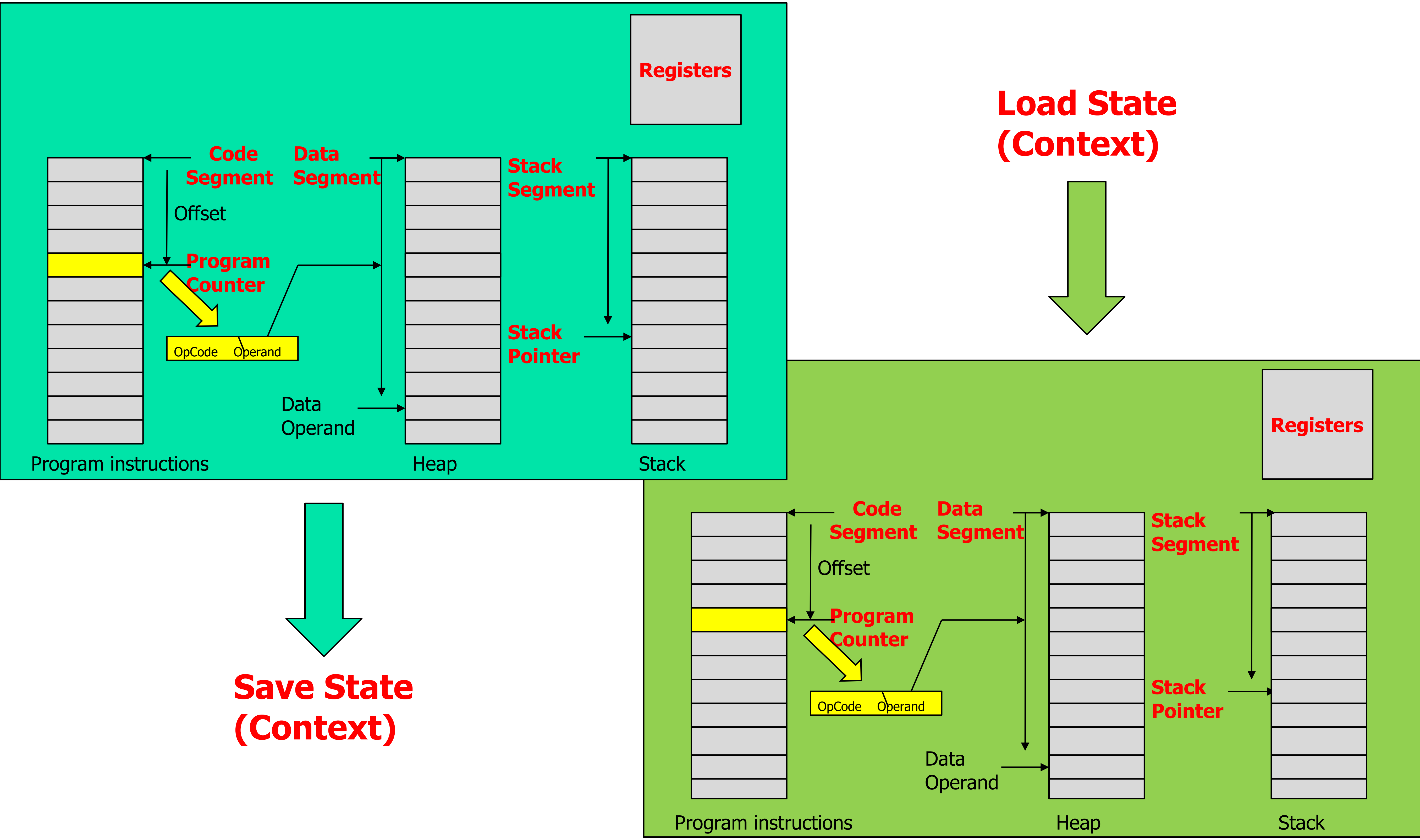


What's a 'real' CPU?

What's the STATE of a real CPU?

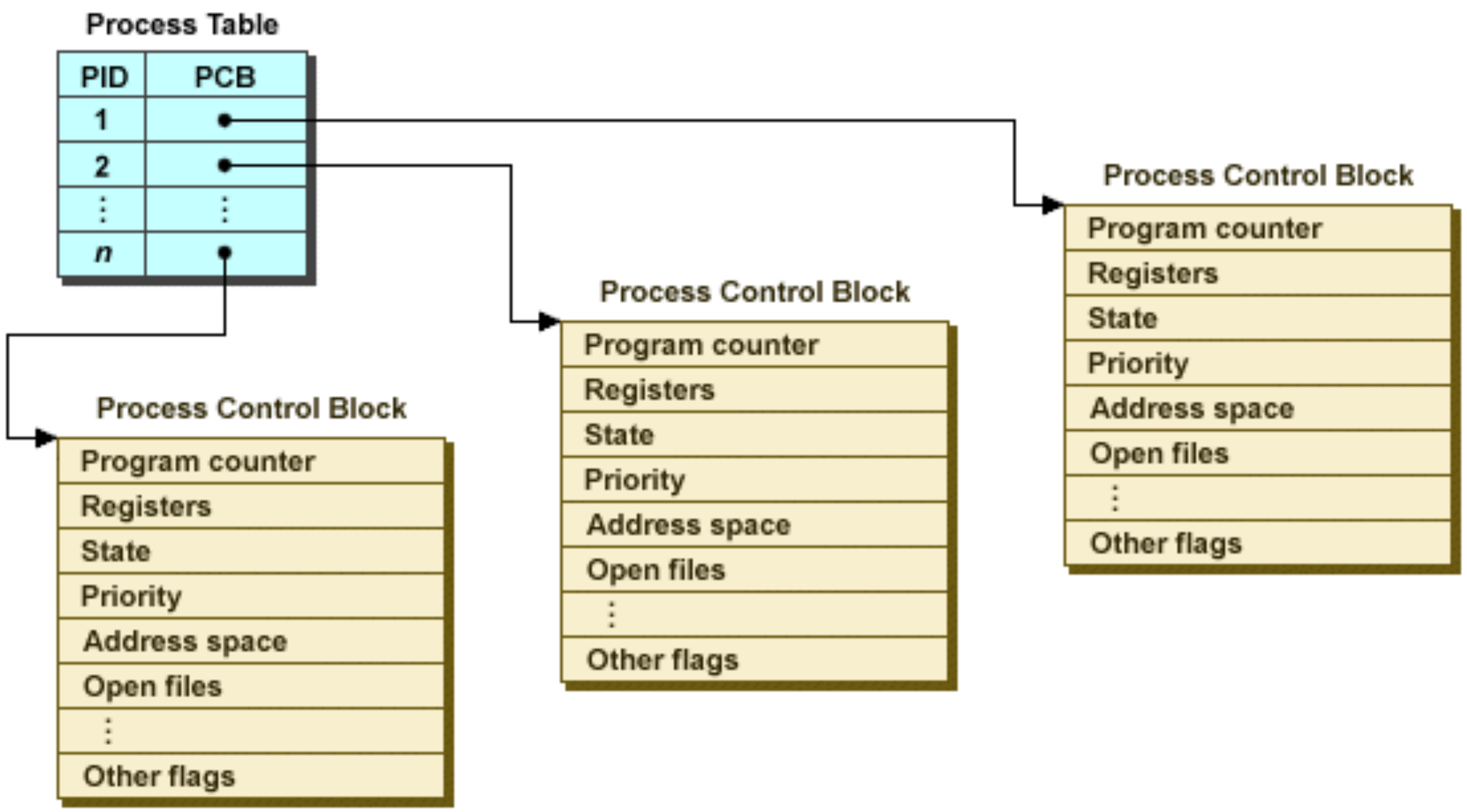


The Context Switch

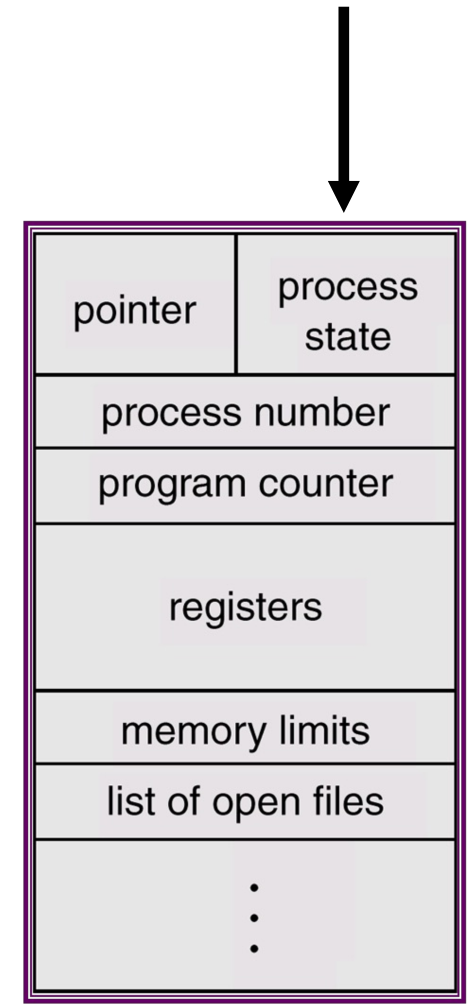


Process Control Block

The state for processes that are not running on the CPU are maintained in the Process Control Block (PCB) data structure

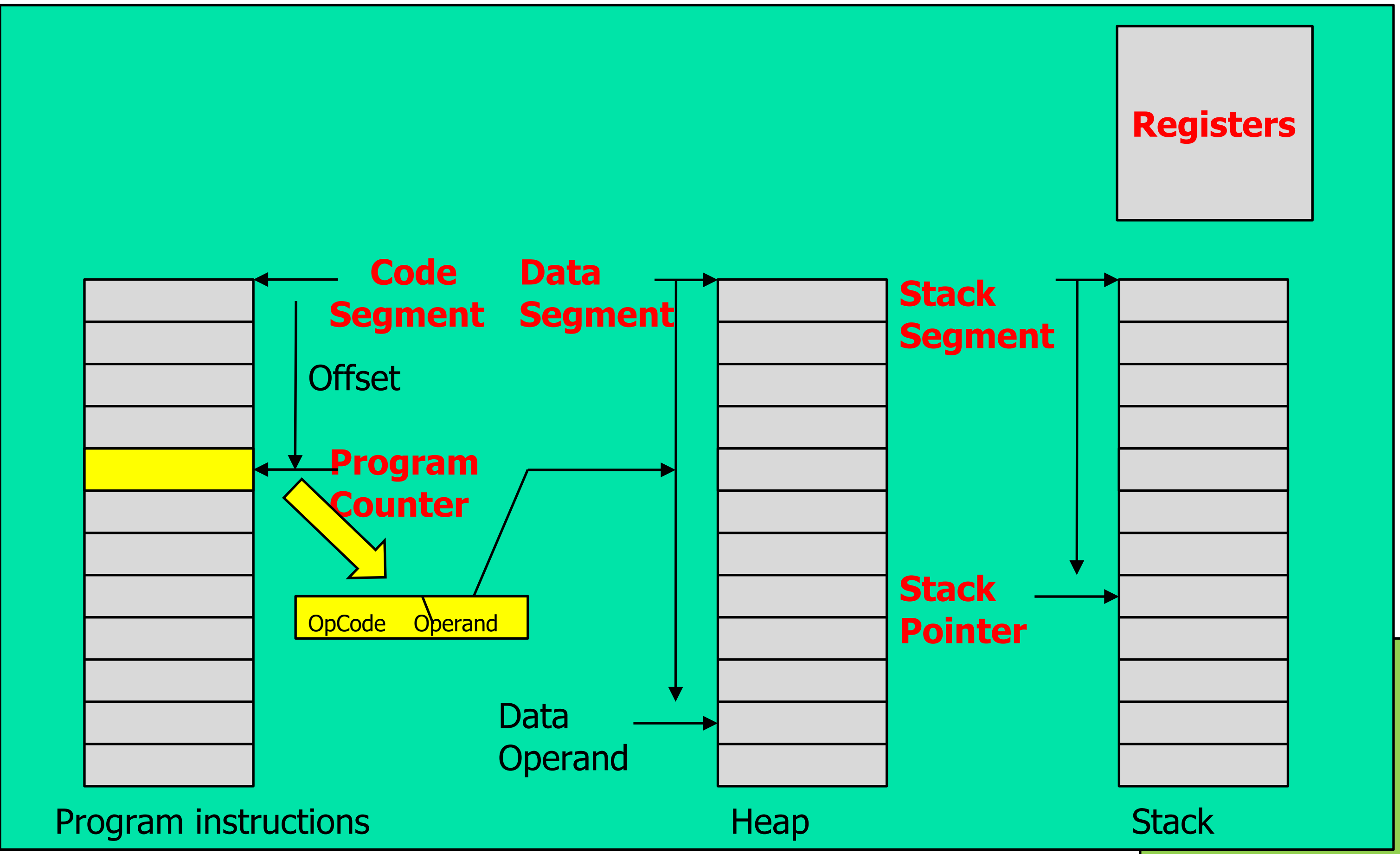


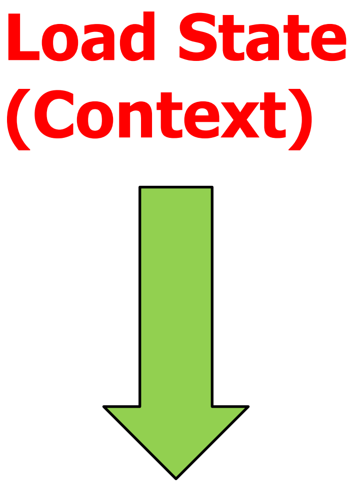
Updated during context switch

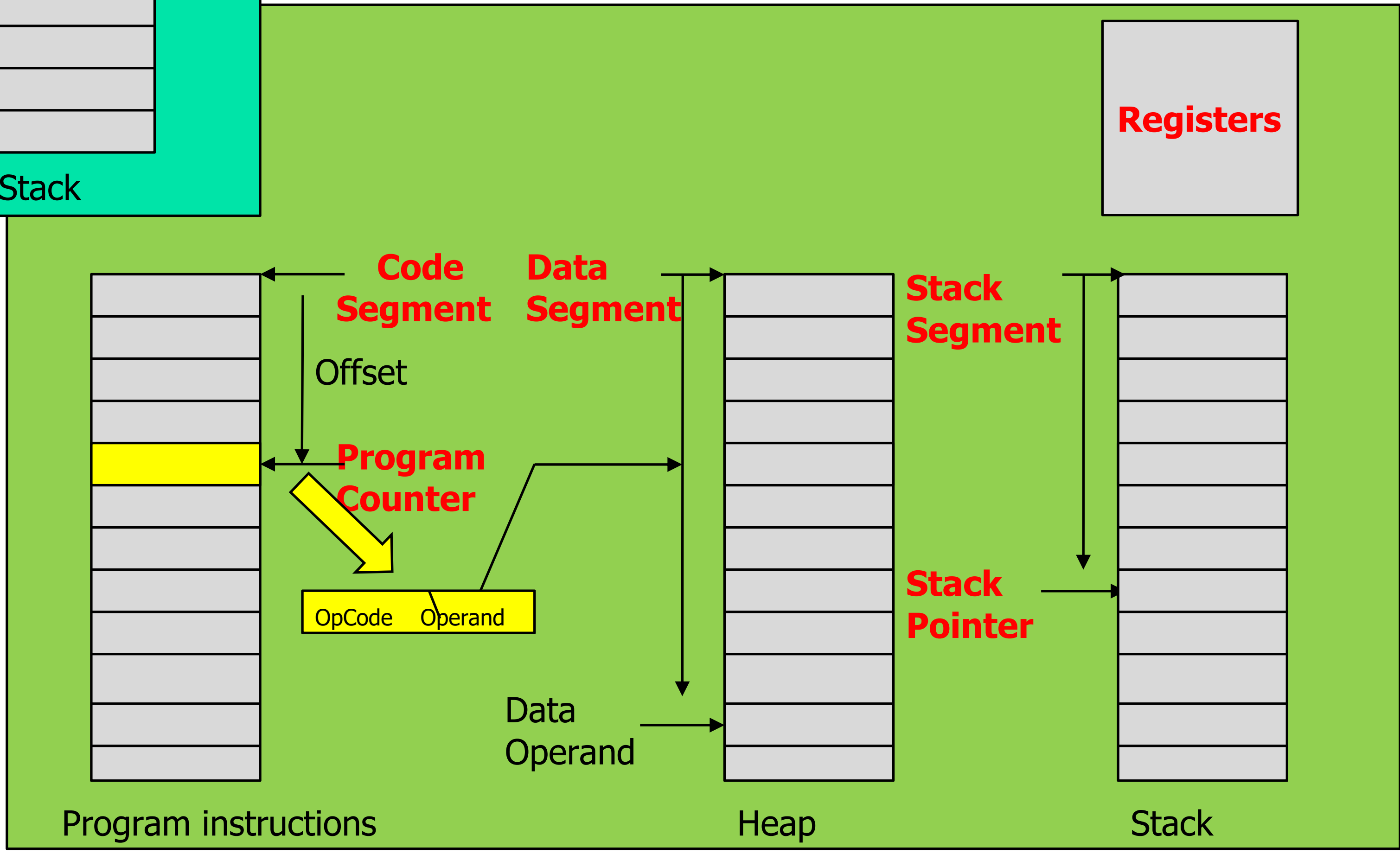


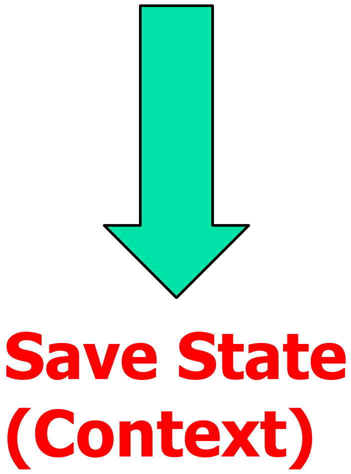
An alternate PCB diagram

The Context Switch



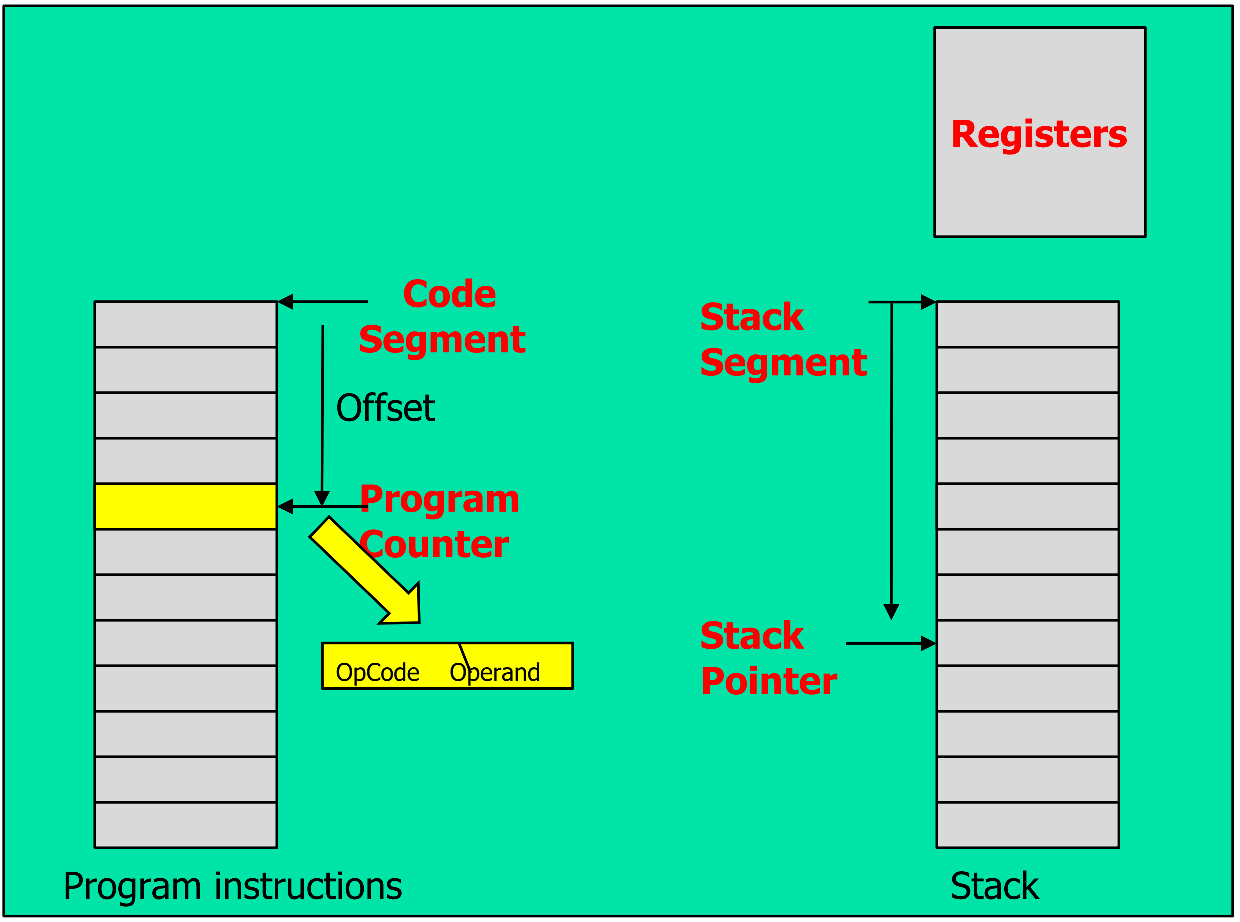


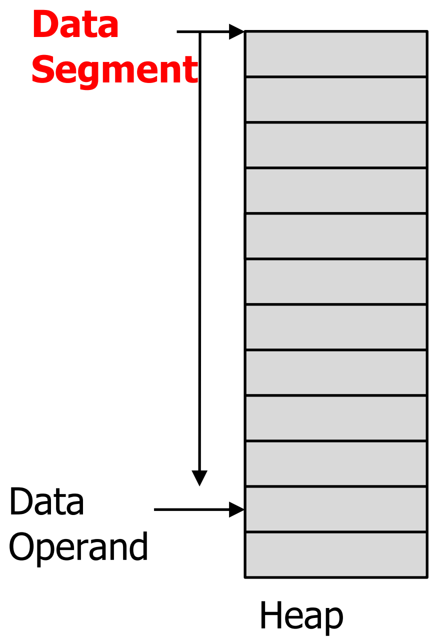




The Context Switch

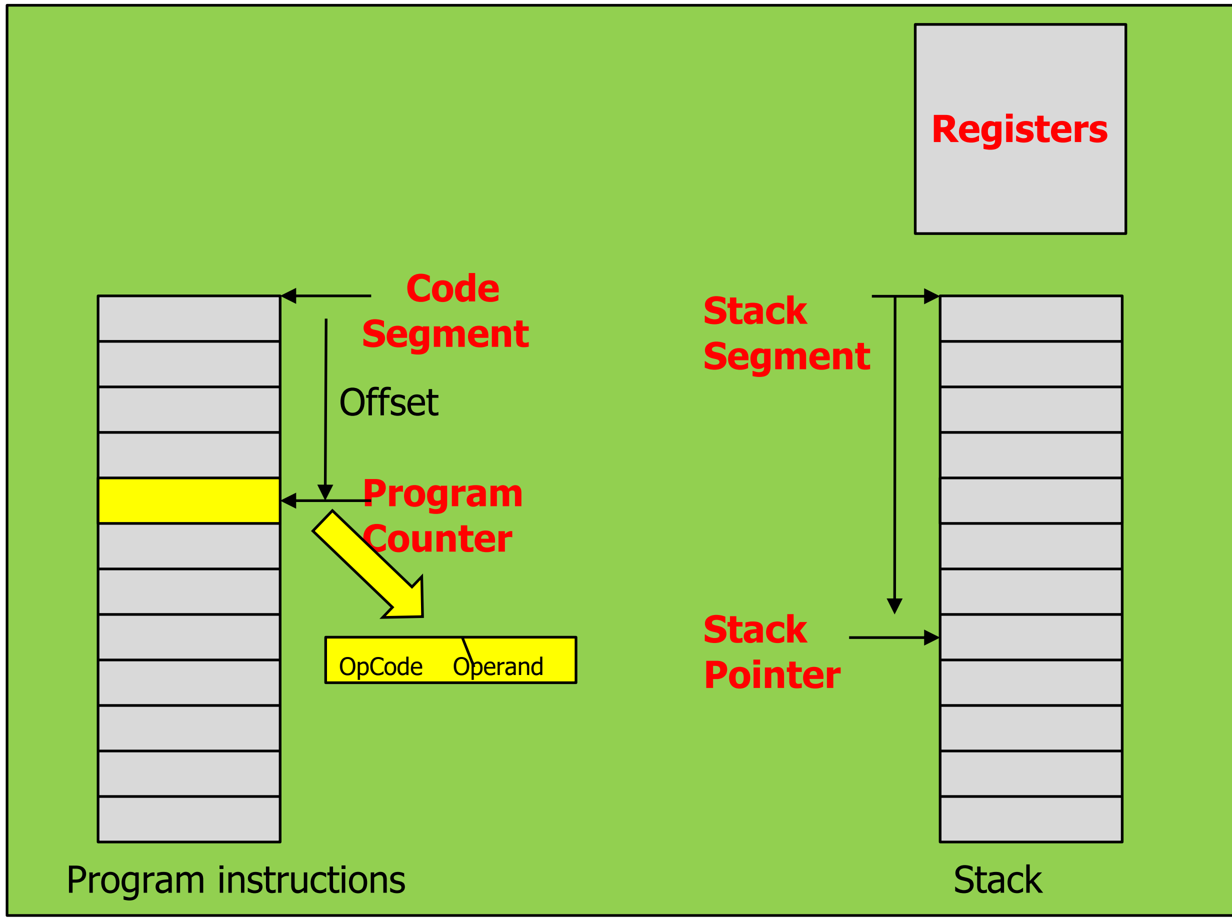
Note: In **thread** context switches, heap is not switched!



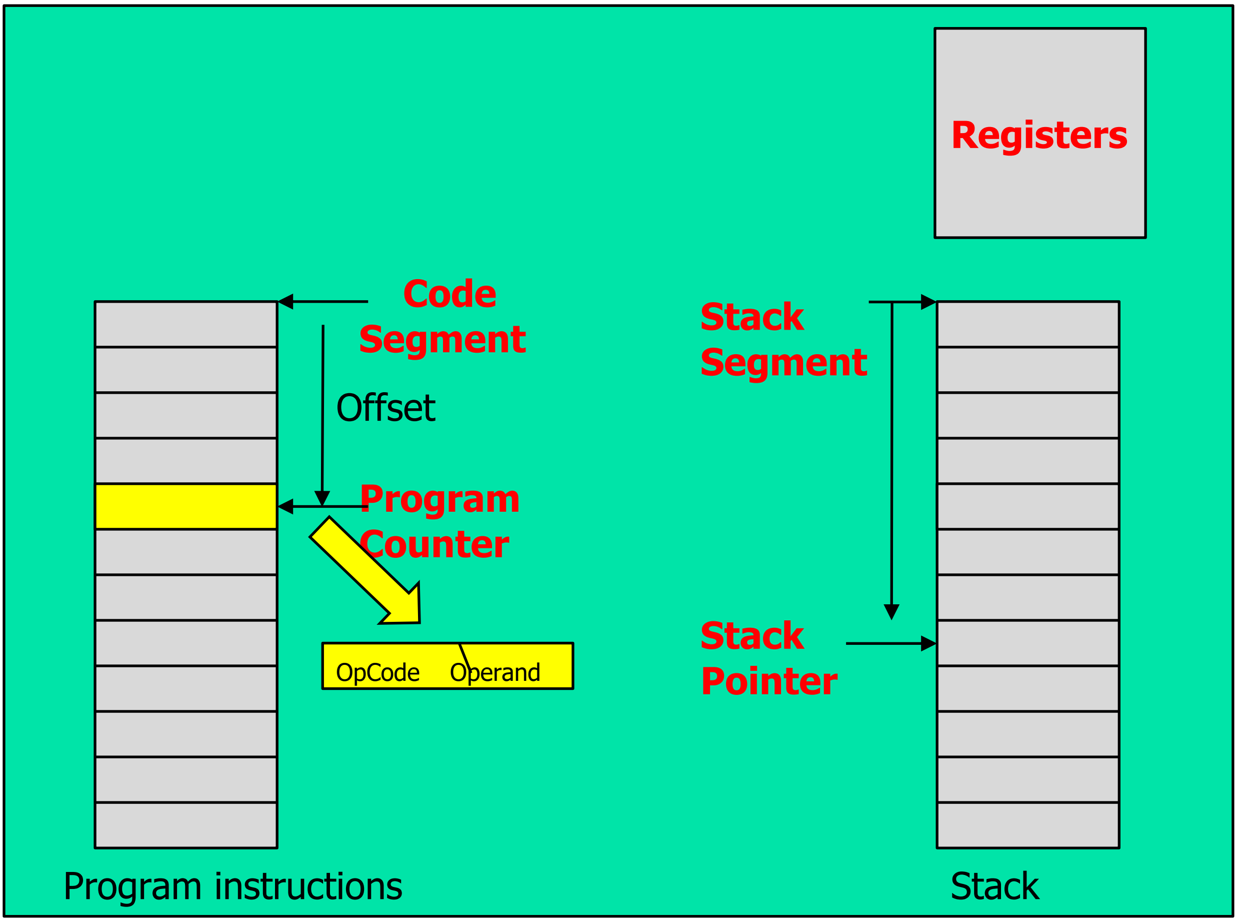


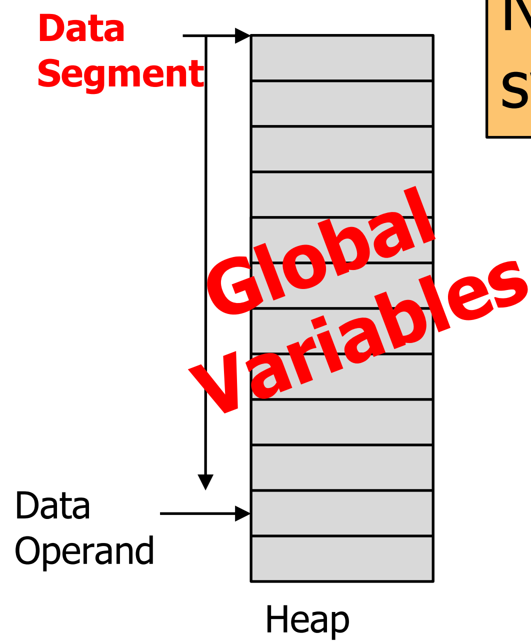
Load State (Context)

Save State (Context)



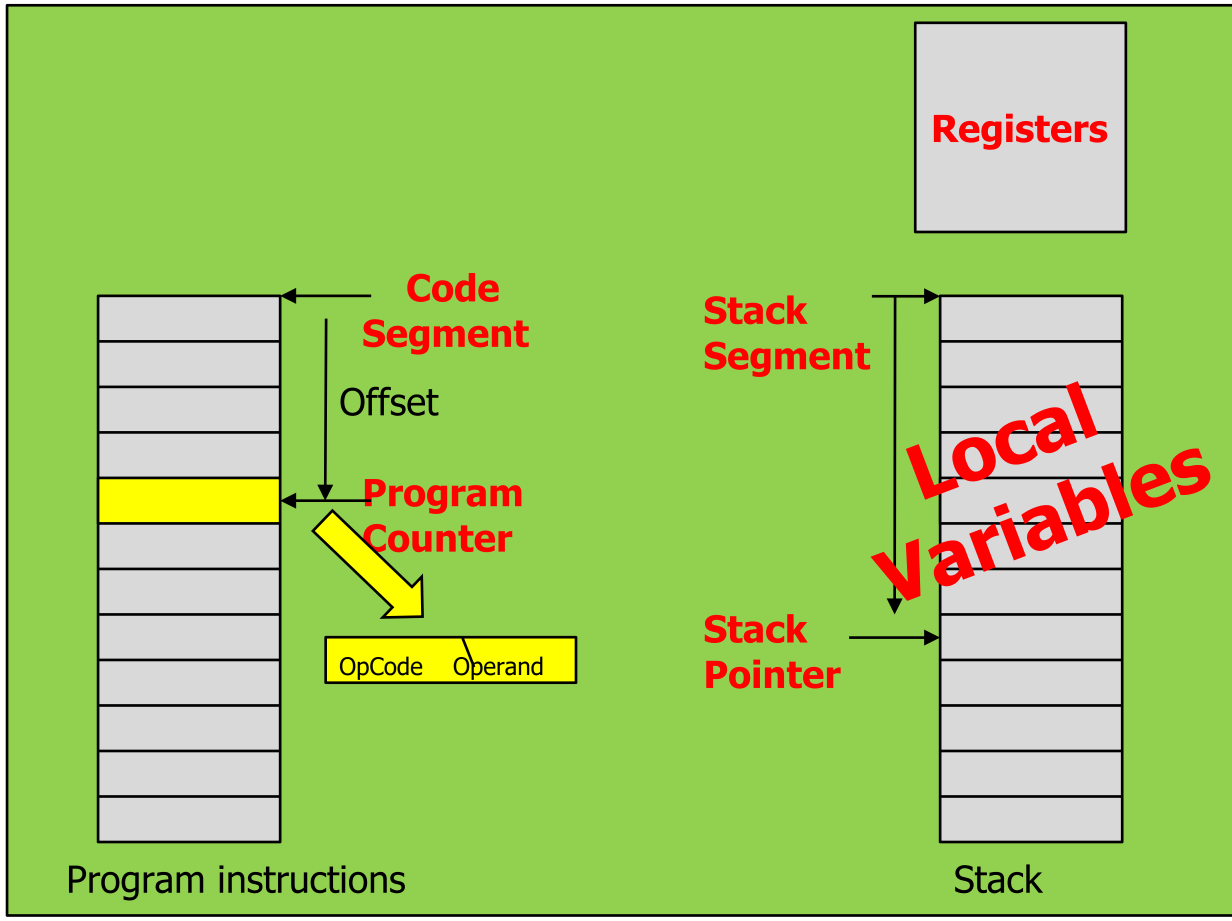
The Context Switch





Note: In **thread** context switches, heap is not switched!

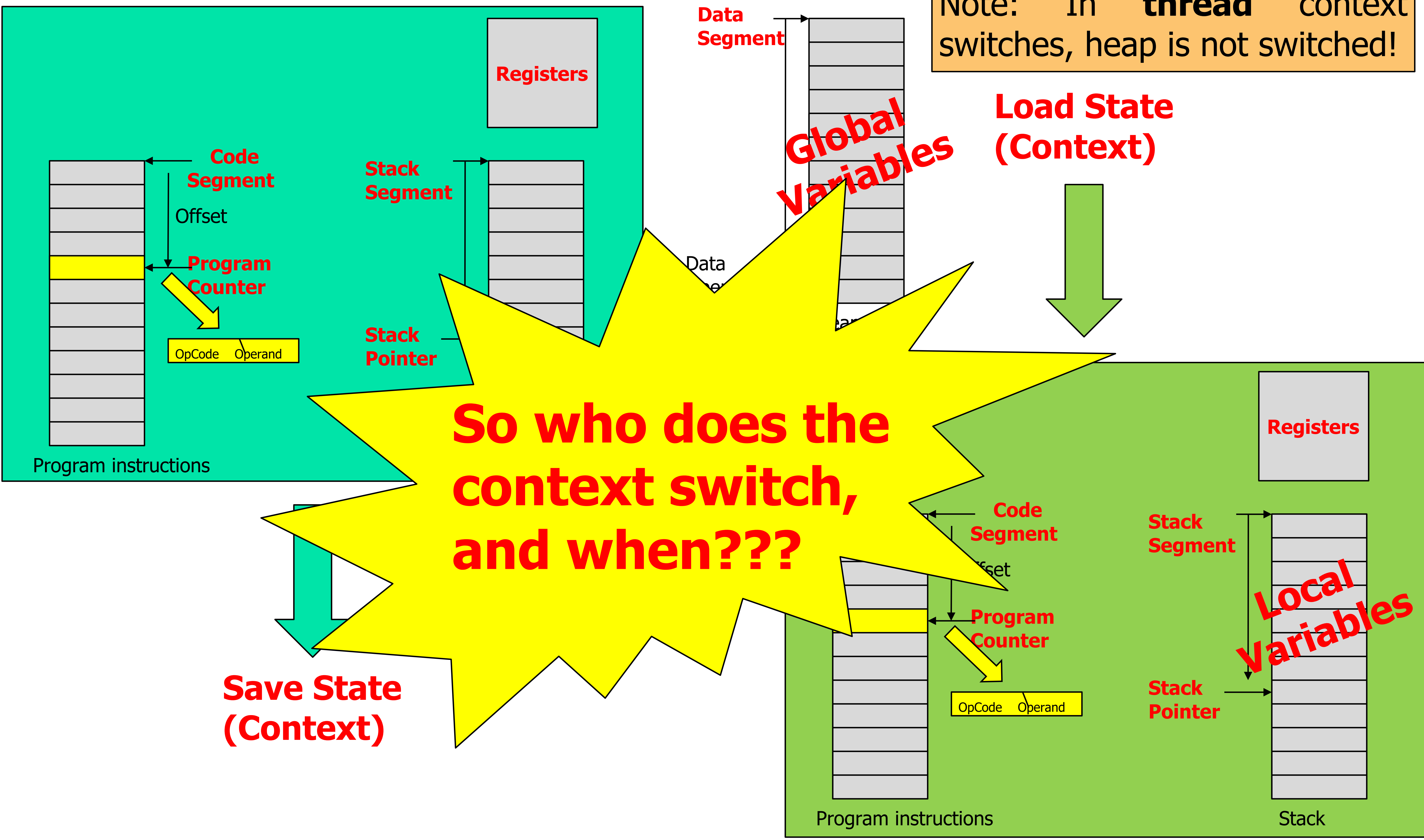
Load State (Context)



Save State (Context)

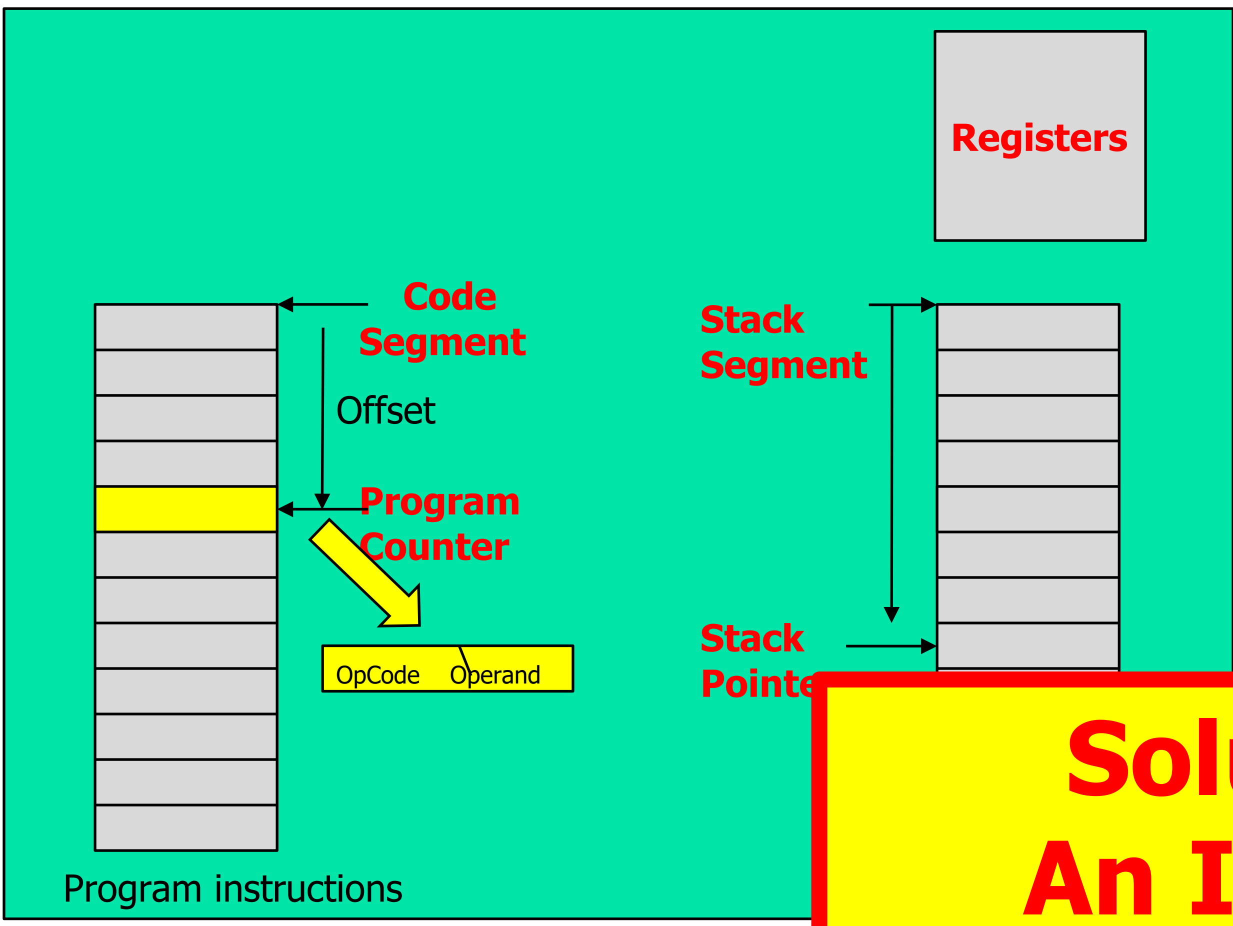
Thread Context Switch

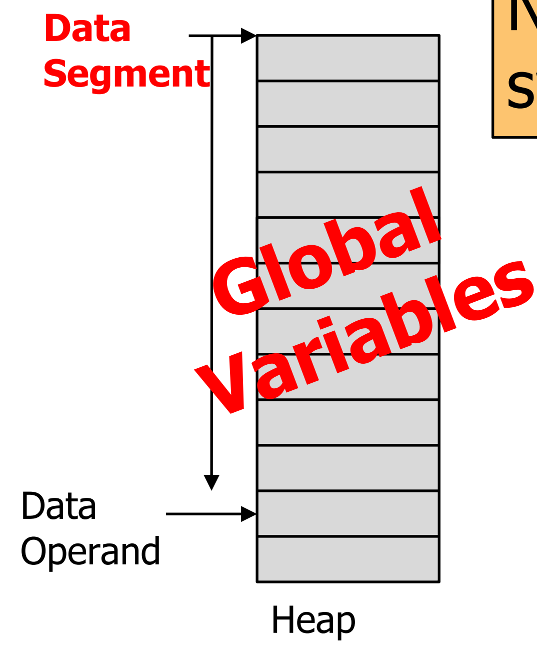
Note: In **thread** context switches, heap is not switched!



Thread Context Switch

Note: In **thread** context switches, heap is not switched!

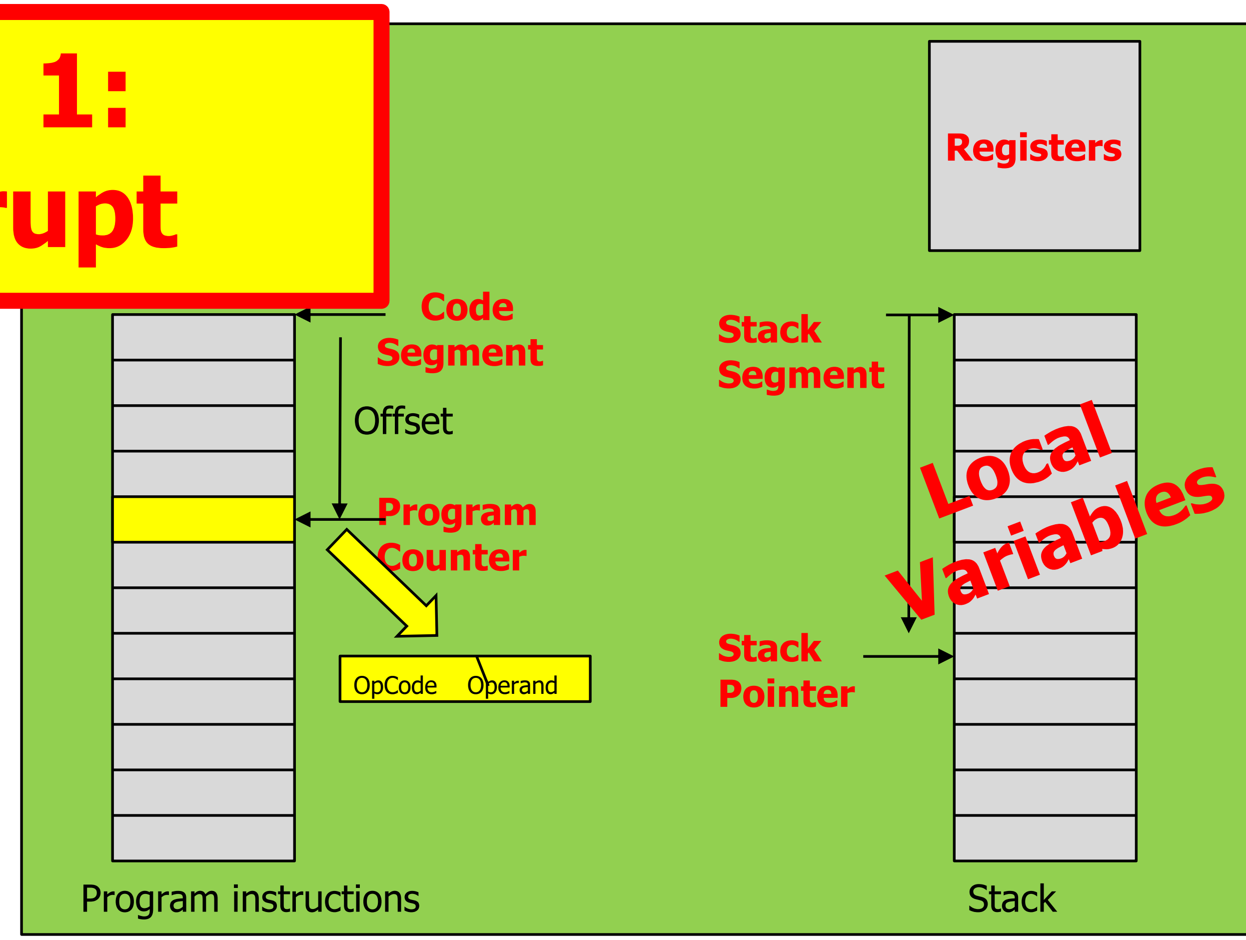




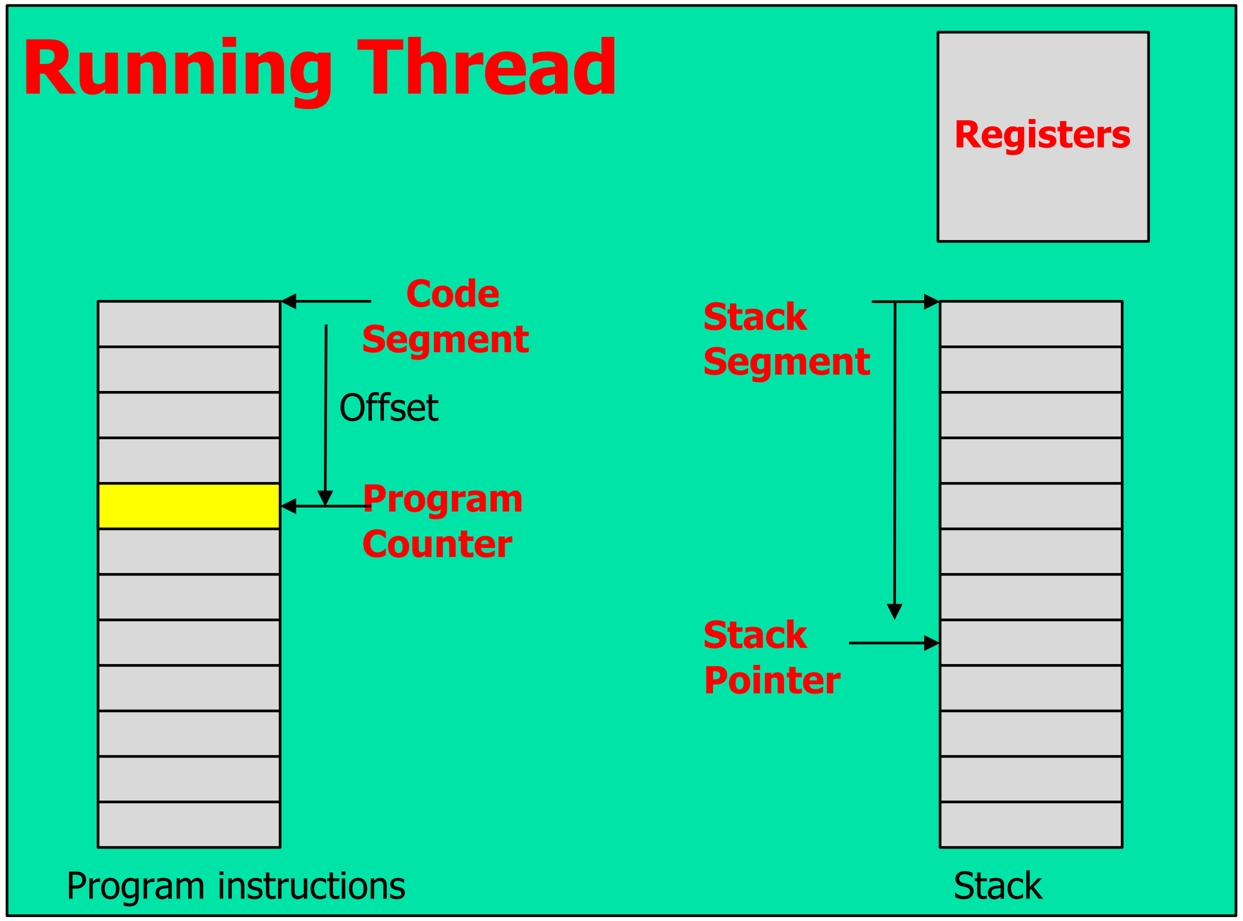
Load State (Context)

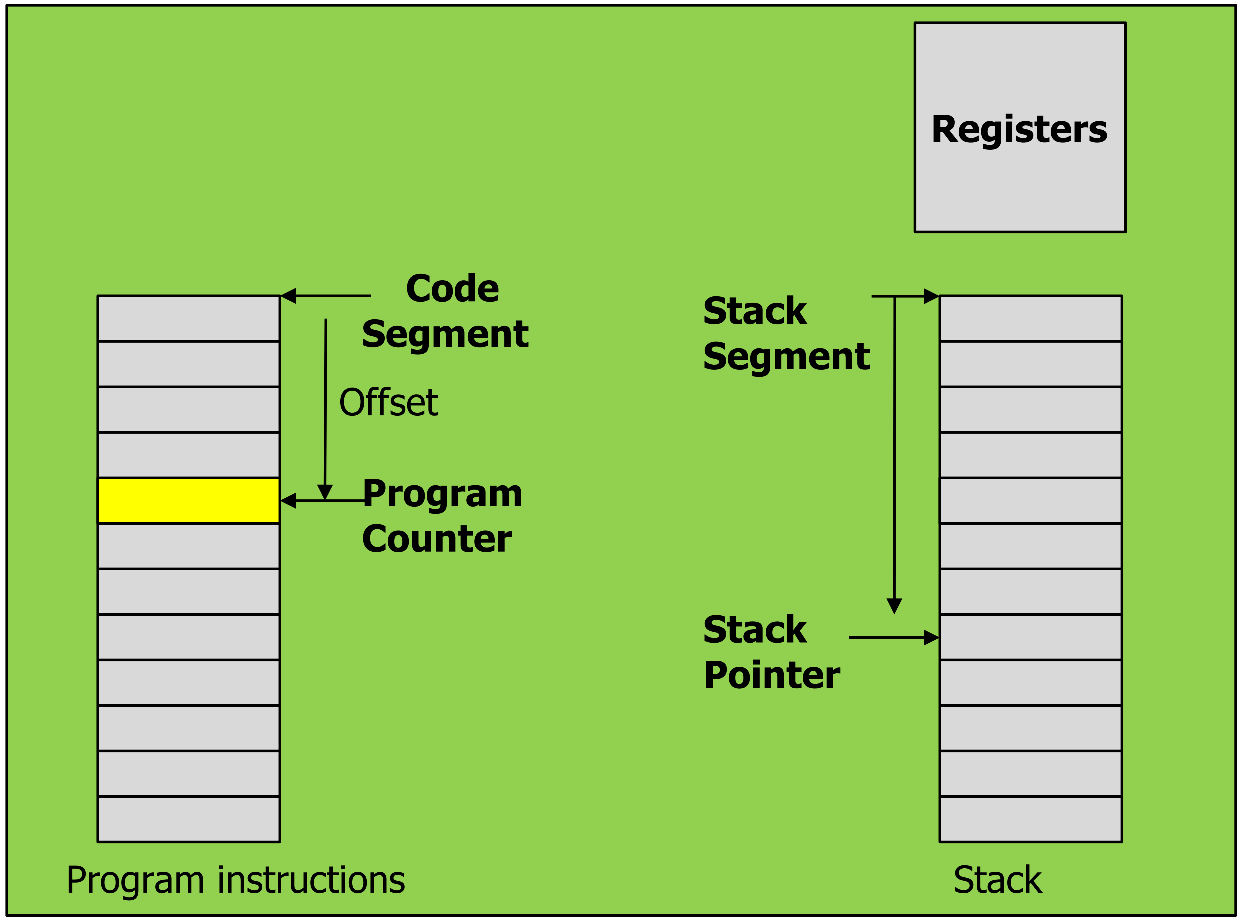
Solution 1: An Interrupt

Save State (Context)

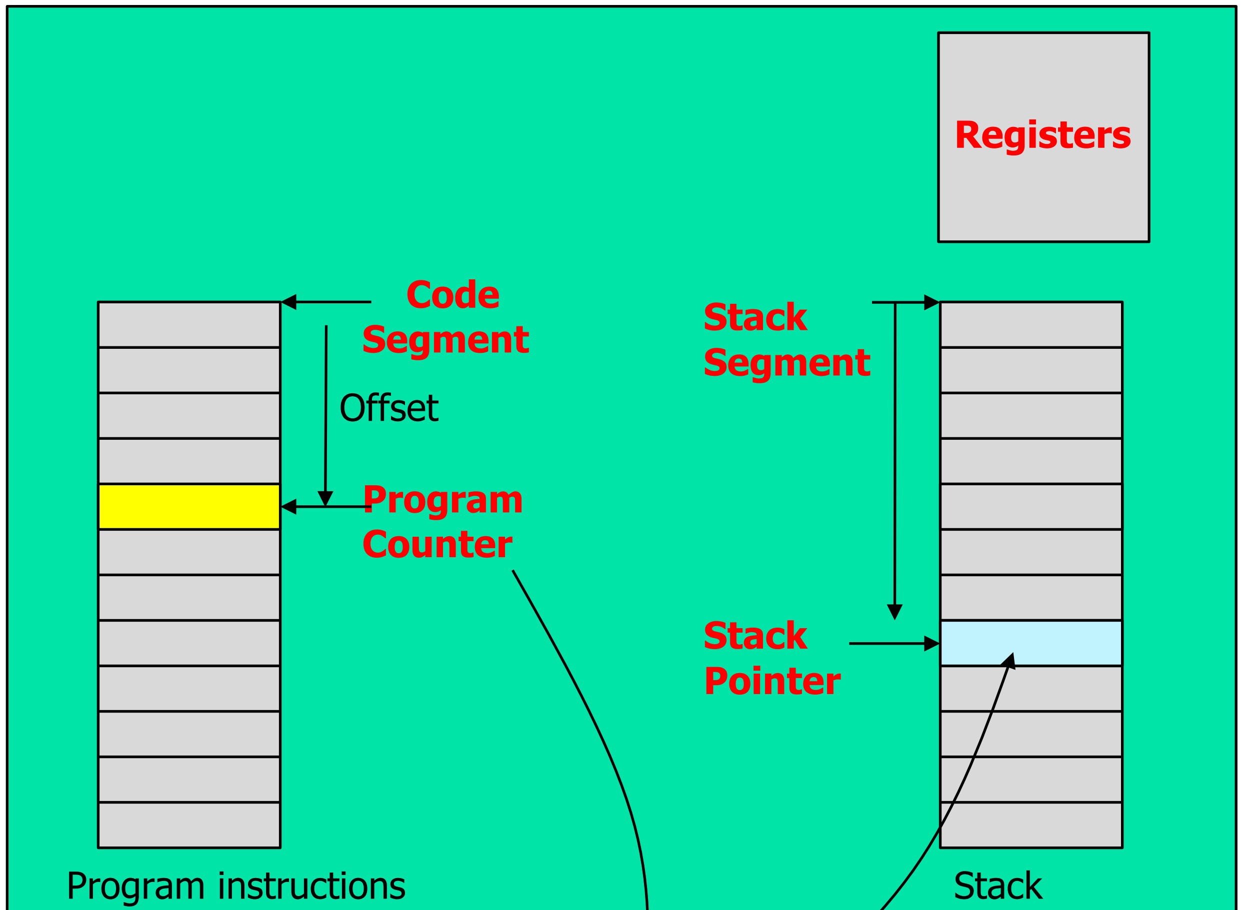


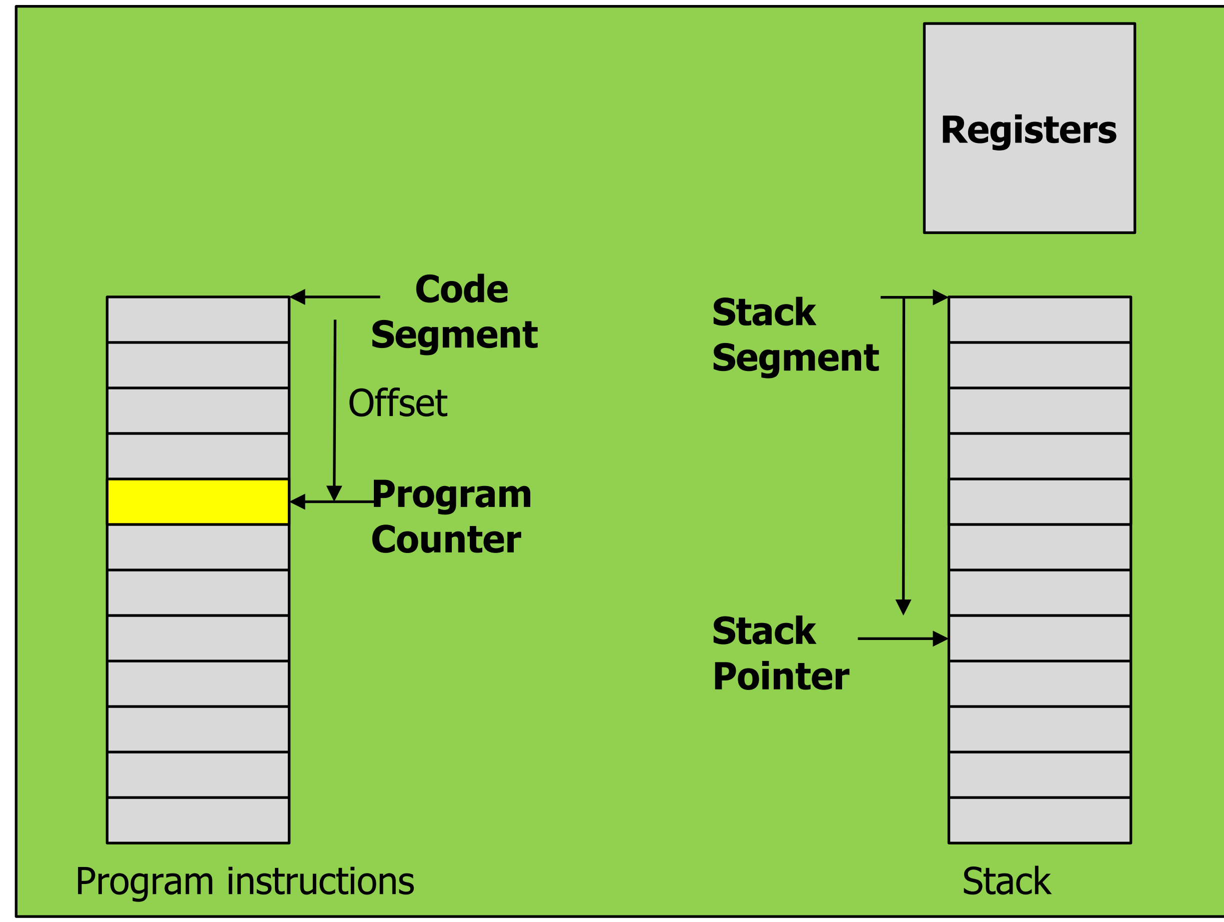
CTX Switch: Interrupt





CTX Switch: Interrupt

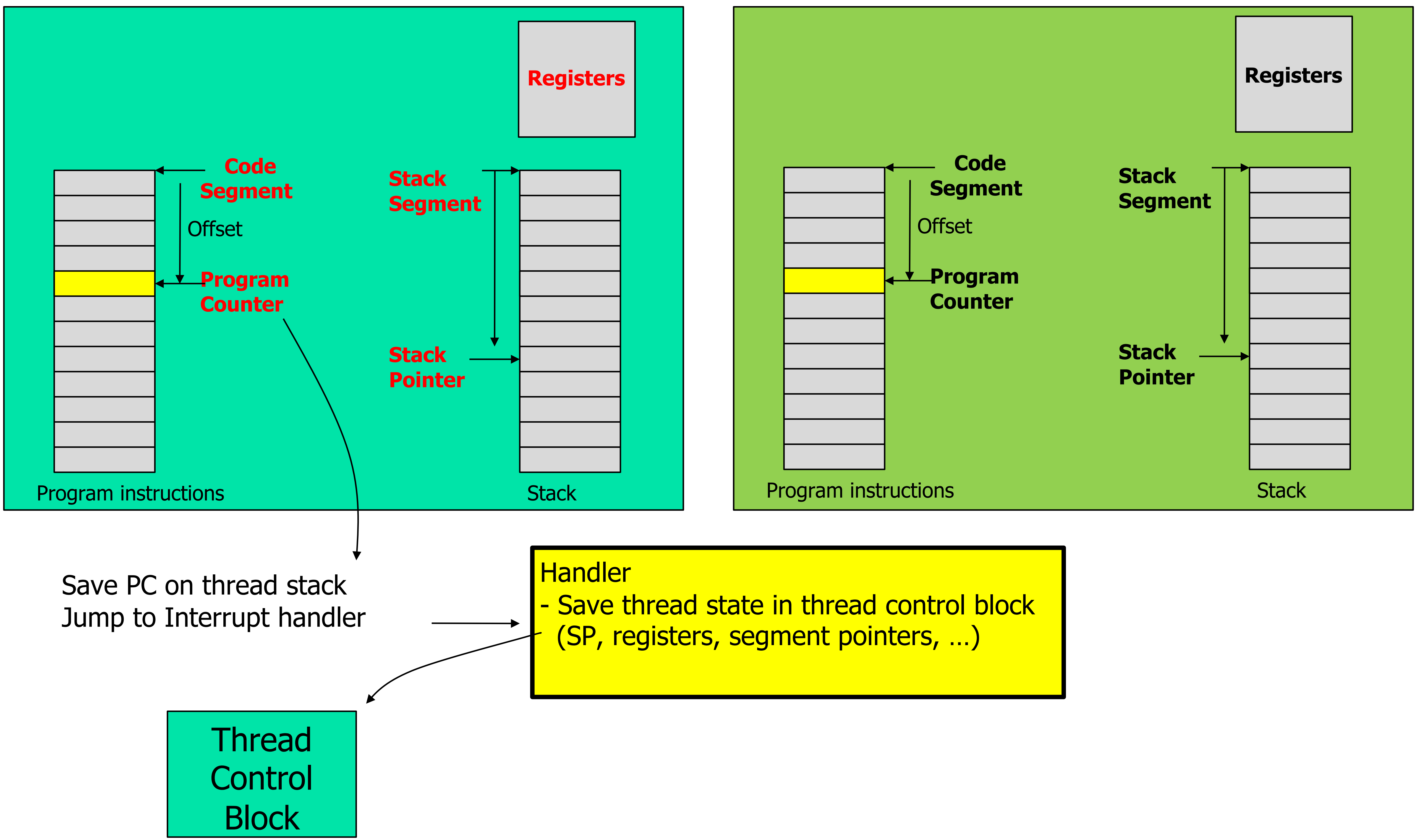




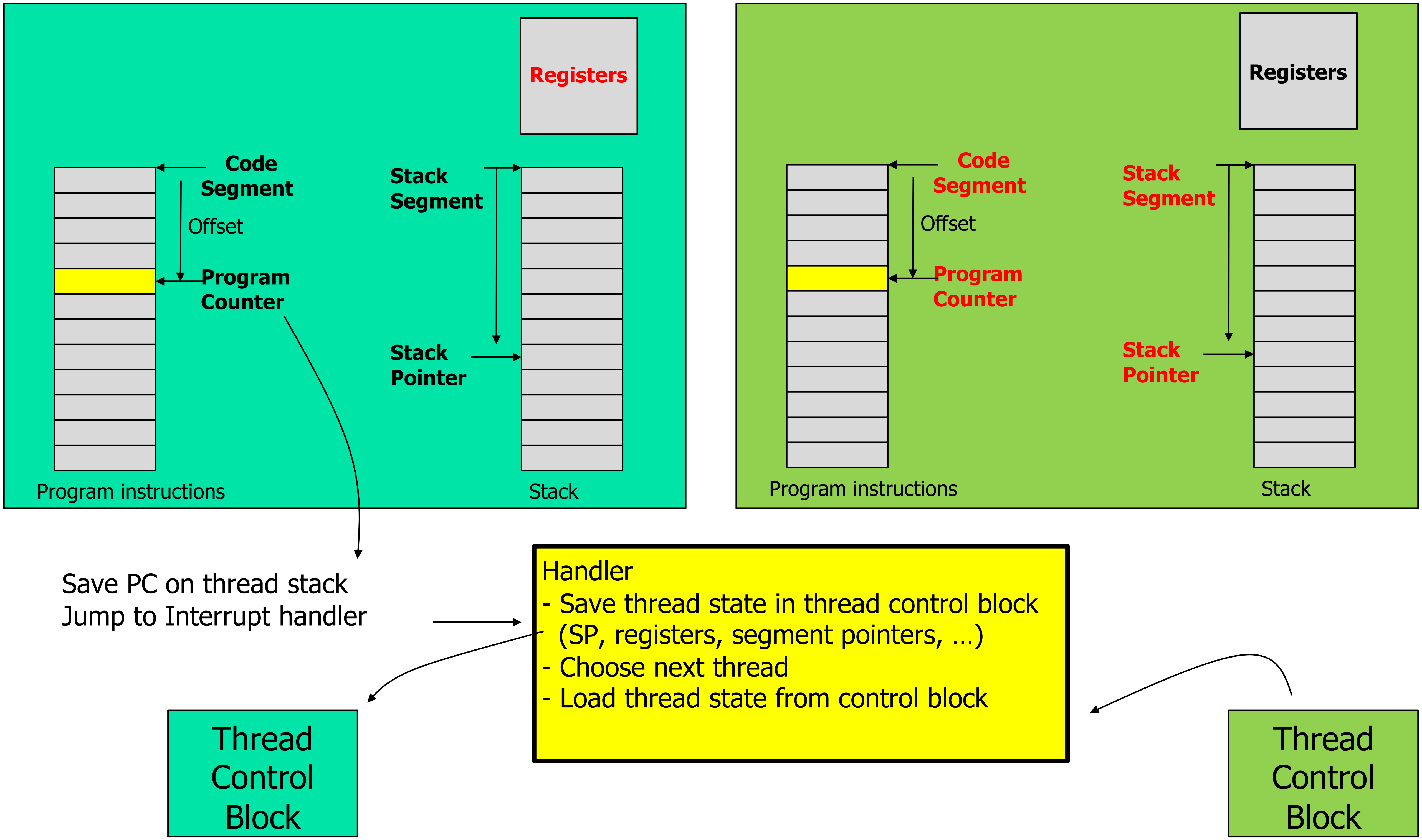
Interrupt

Save PC on thread stack
Jump to Interrupt handler

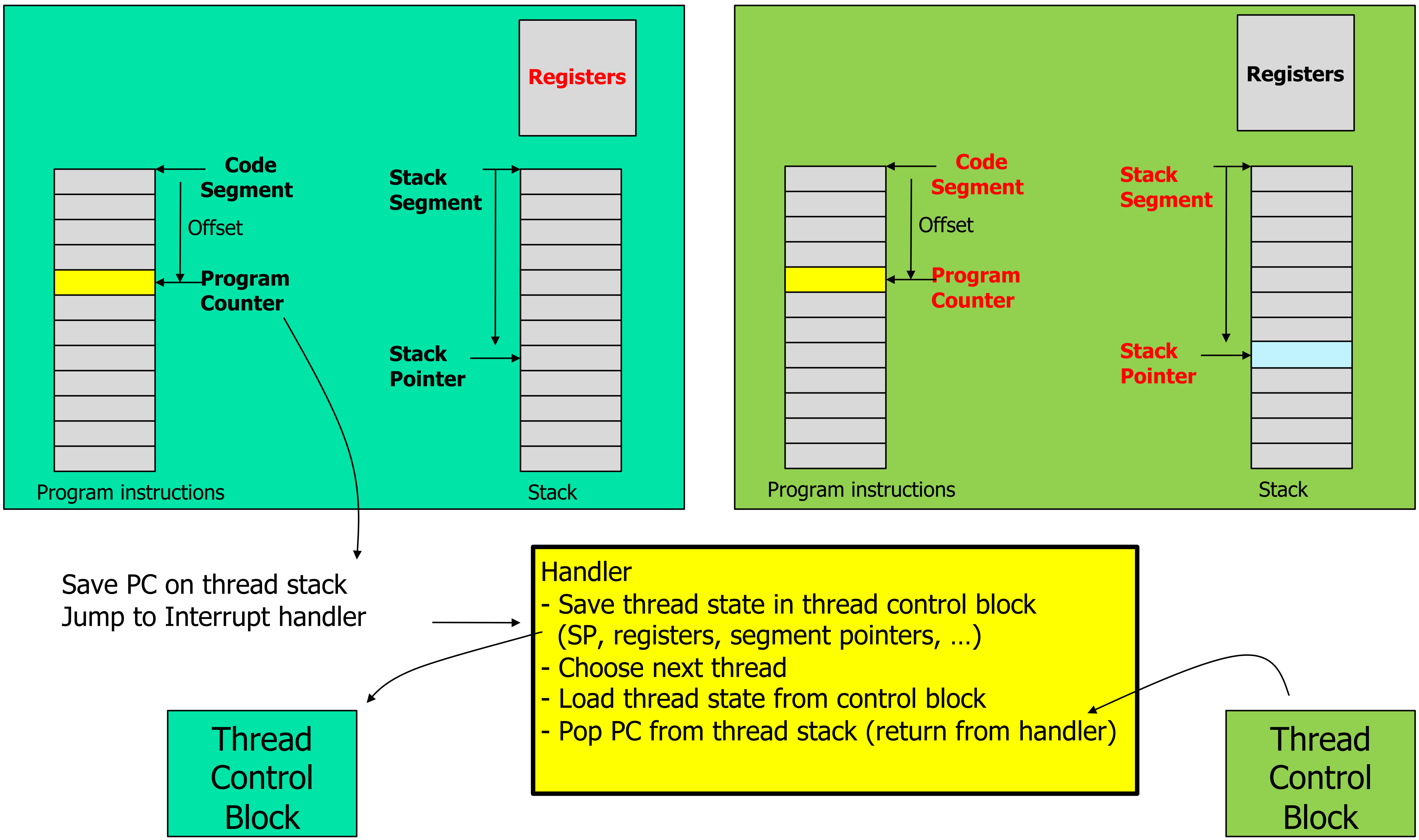
CTX Switch: Interrupt



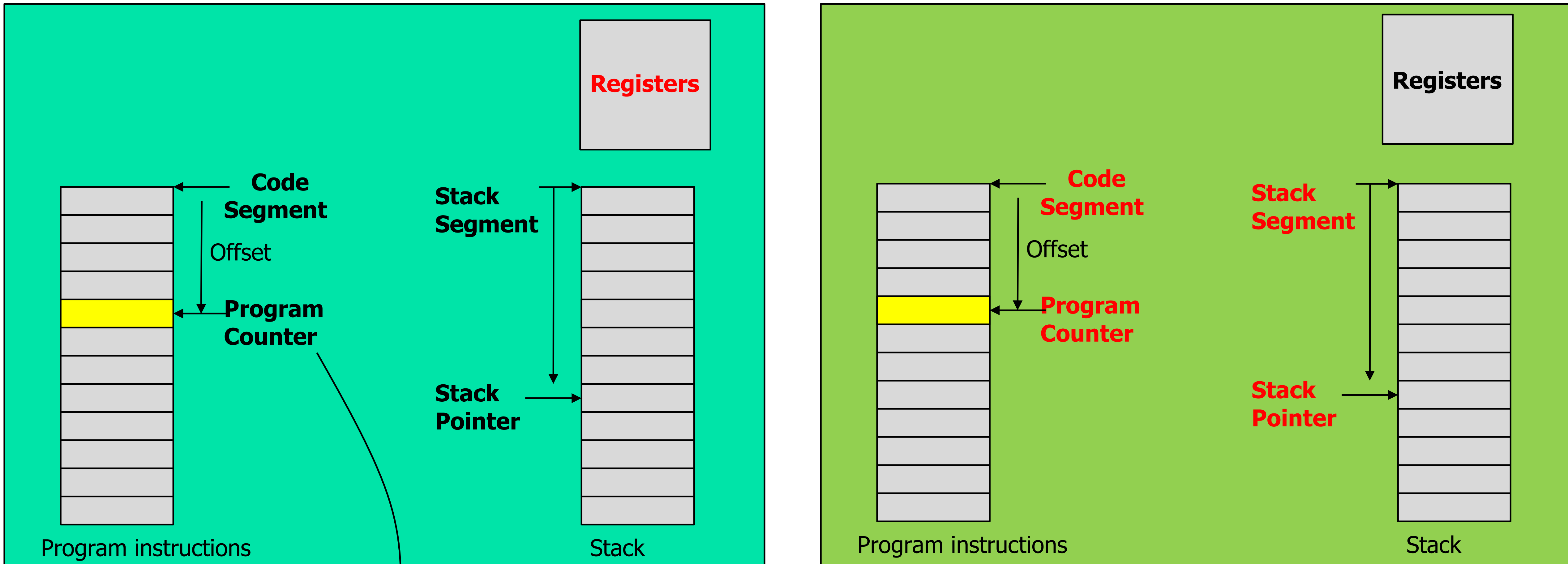
CTX Switch: Interrupt



CTX Switch: Interrupt



CTX Switch: Interrupt



Save PC on thread stack
Jump to Interrupt handler

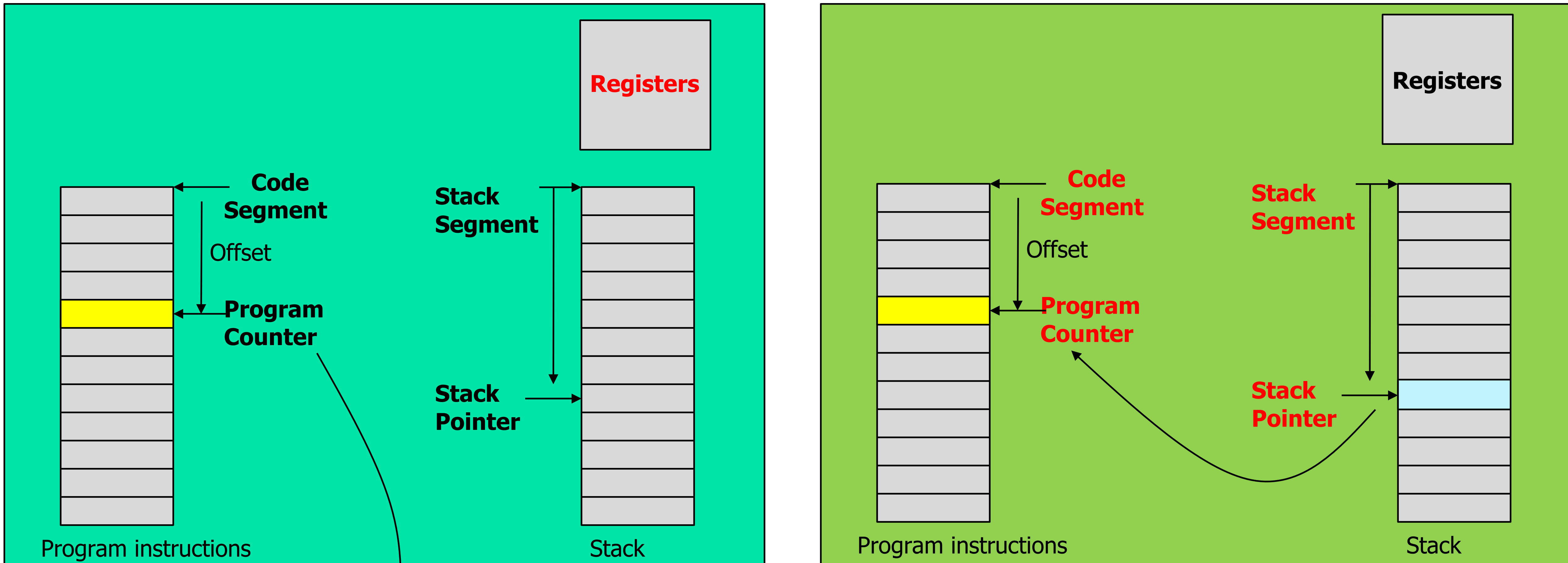
Handler
- Save thread state in thread control block
  (SP, registers, segment pointers, …)
- Choose next thread
- Load thread state from control block
- Pop PC from thread stack (return from handler)

Thread Control Block

Thread Control Block

Where does it return?

CTX Switch: Interrupt



Save PC on thread stack
Jump to Interrupt handler

Handler
- Save thread state in thread control block
  (SP, registers, segment pointers, …)
- Choose next thread
- Load thread state from control block
- Pop PC from thread stack (return from handler)

Thread Control Block

Thread Control Block

Where does it return?

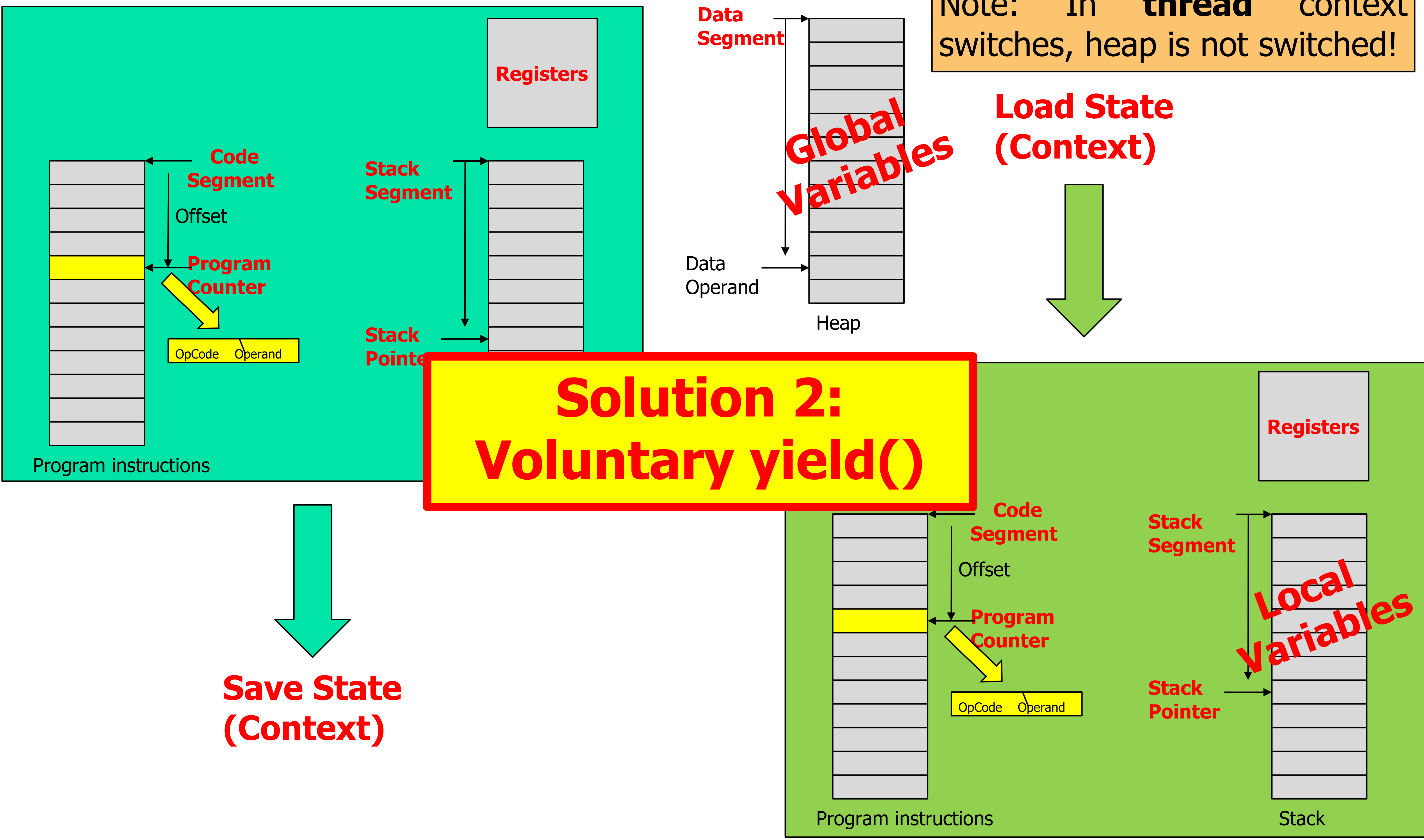
CTX Switch: Interrupt

What are some examples of context switches due to interrupts?

- **Clock Interrupt:** Task exceeds its time slice
- **I/O Interrupt**: Waiting processes may be preempted
- **Memory Fault:** CPU attempts to access a virtual memory address that is not in main memory. OS may resume execution of another process while retrieving the block, then moves process to ready state.

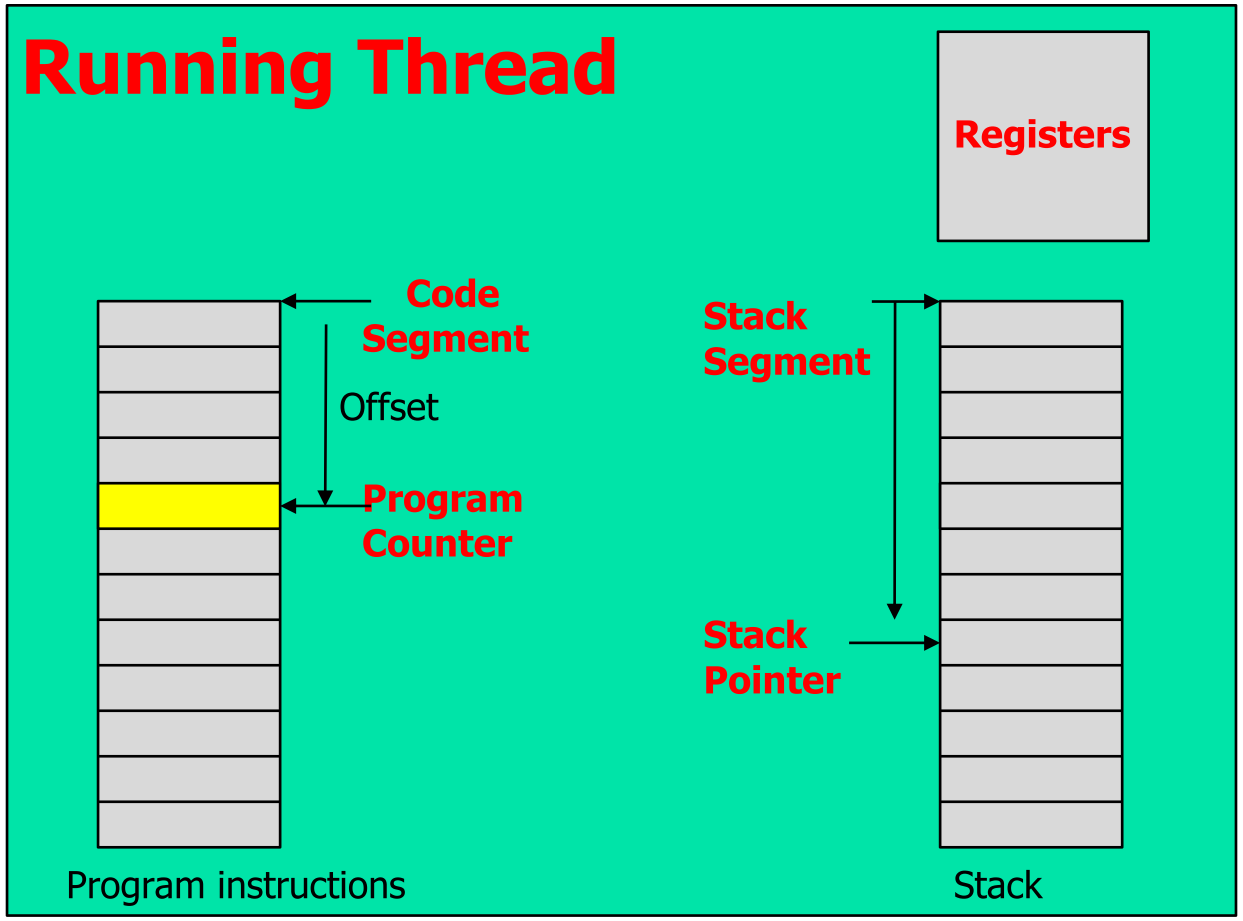
Thread Context Switch

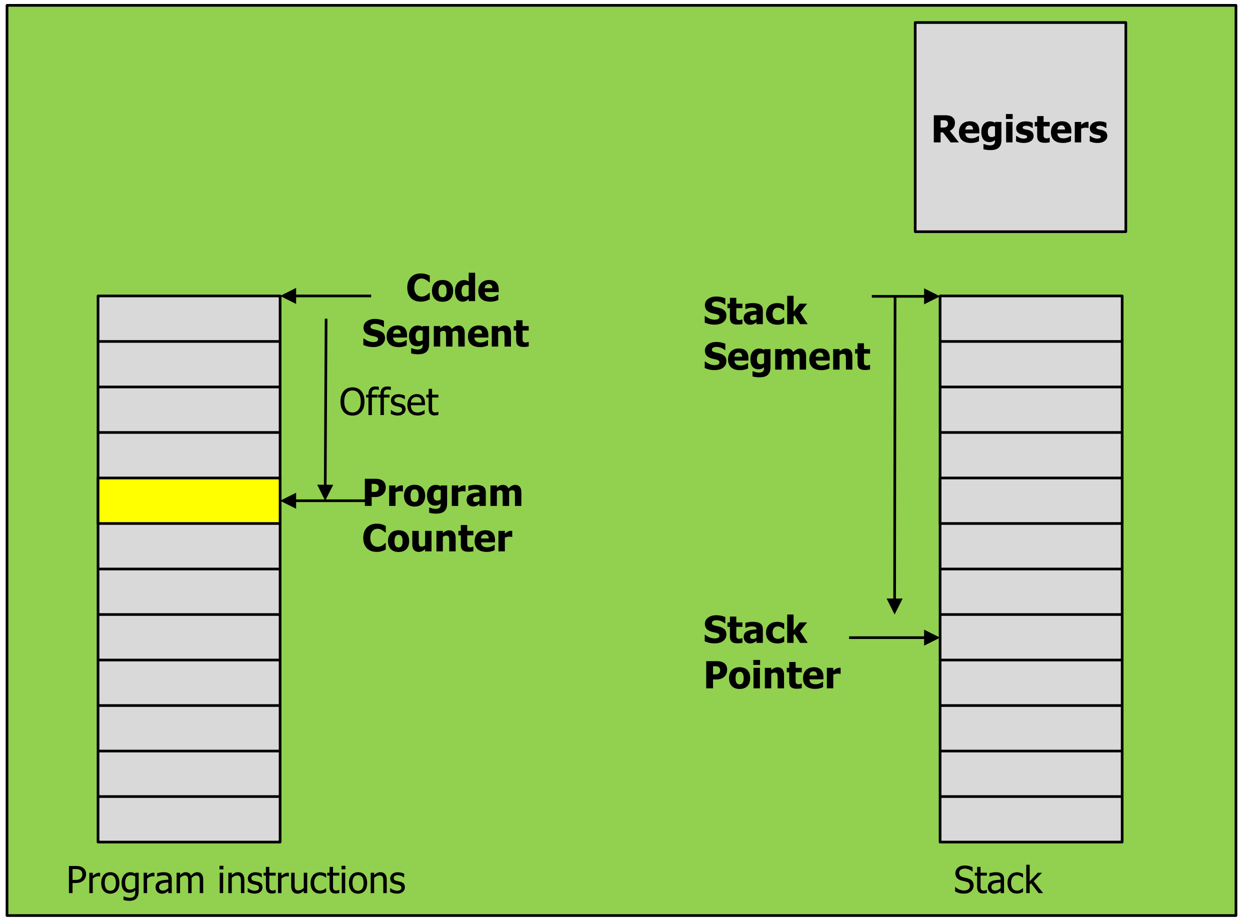
Note: In **thread** context switches, heap is not switched!



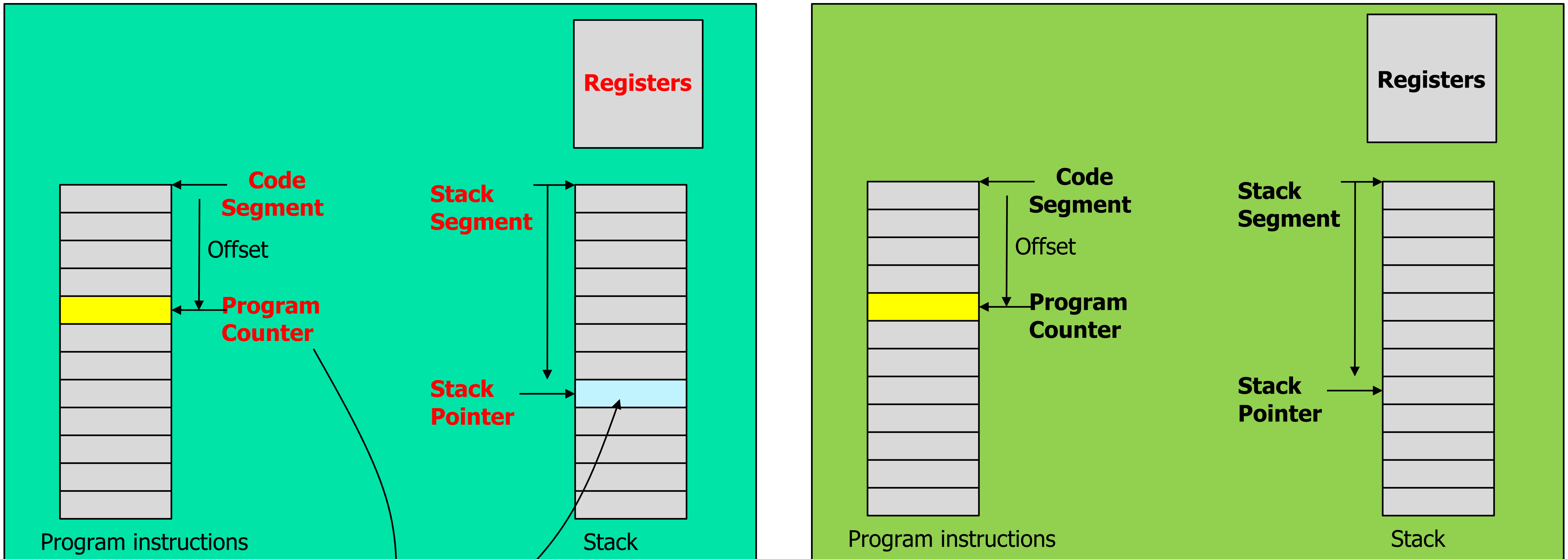
Solution 2: Voluntary yield()

CTX Switch: Yield





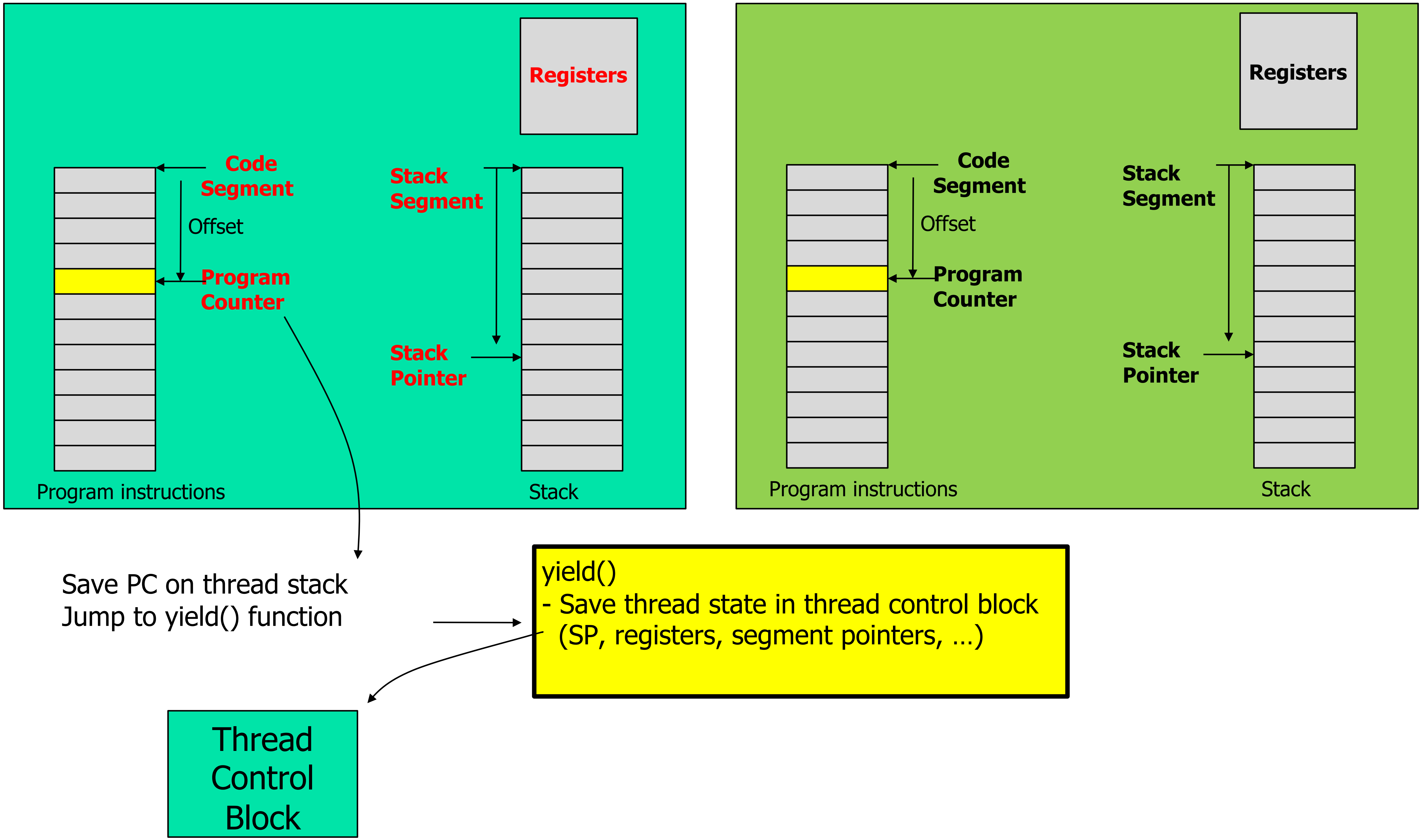
CTX Switch: Yield



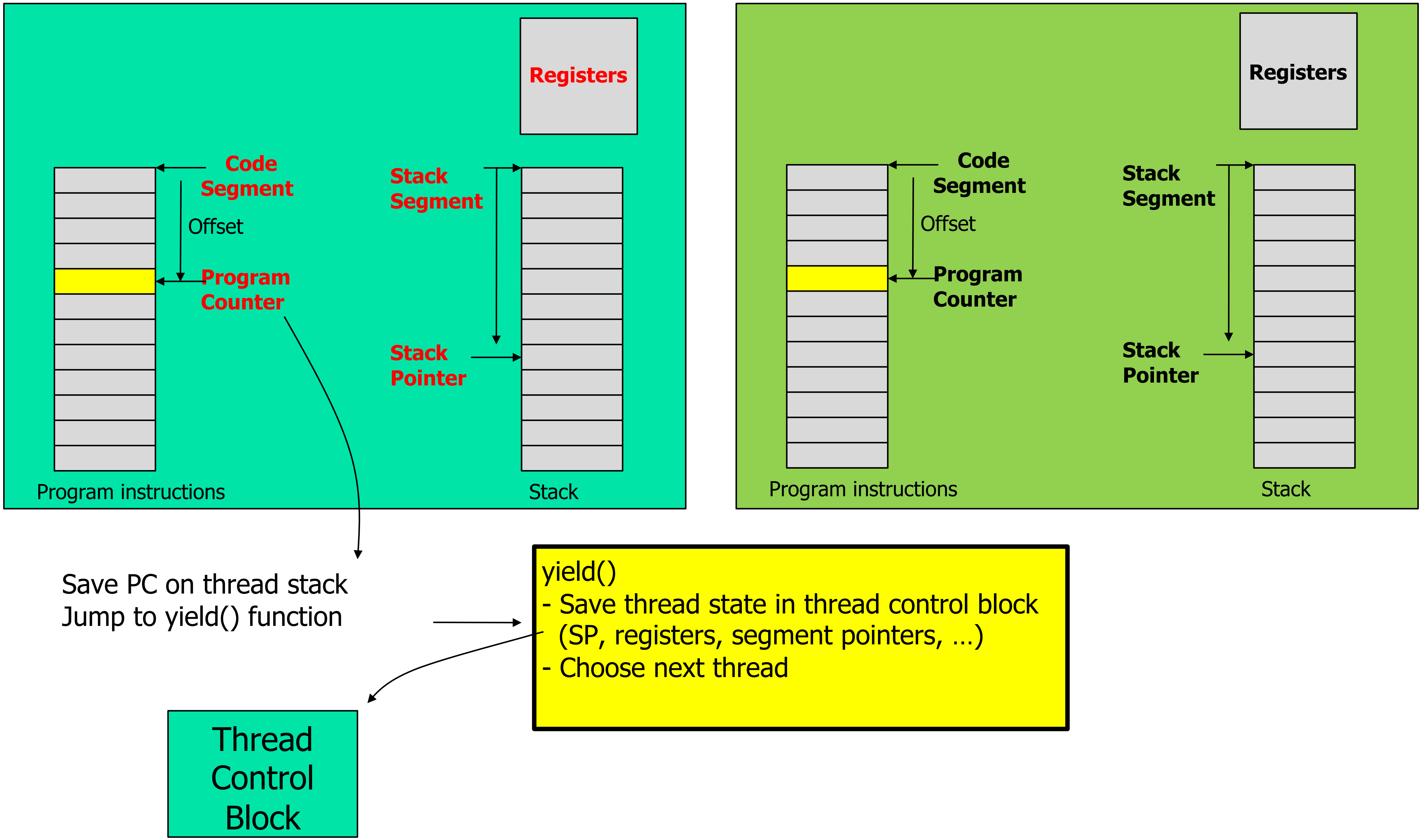
yield()

Save PC on thread stack
Jump to yield() function

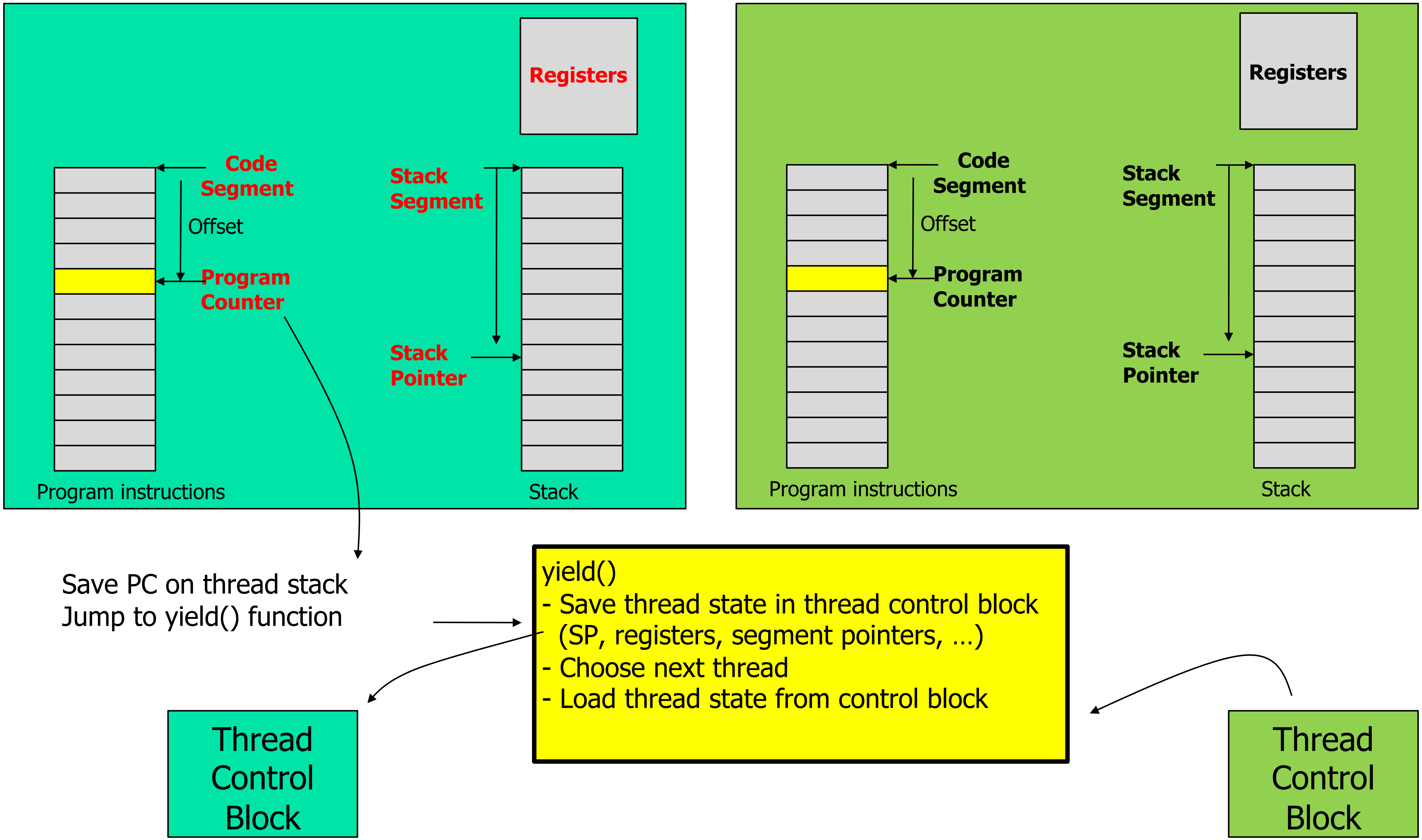
CTX Switch: Yield



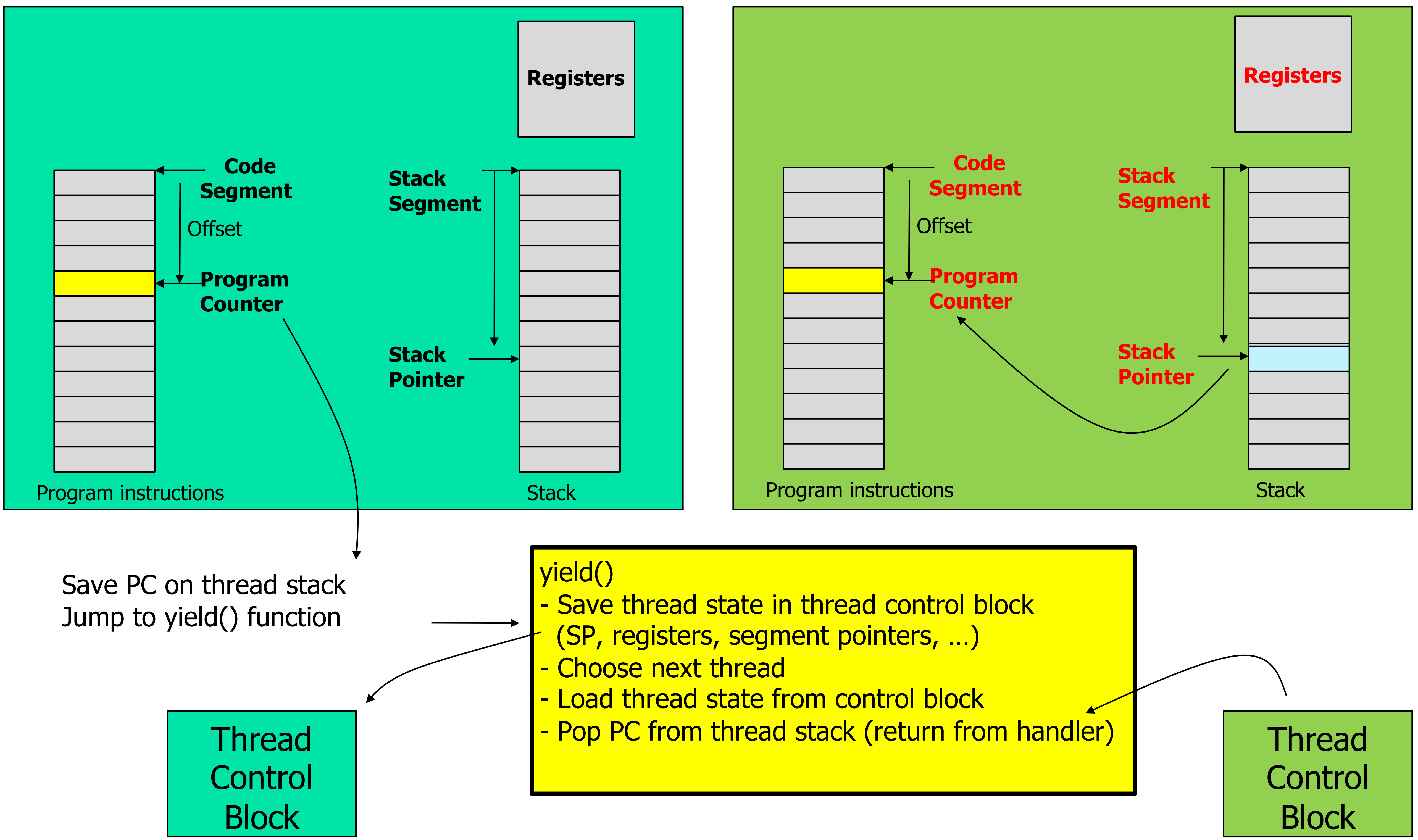
CTX Switch: Yield



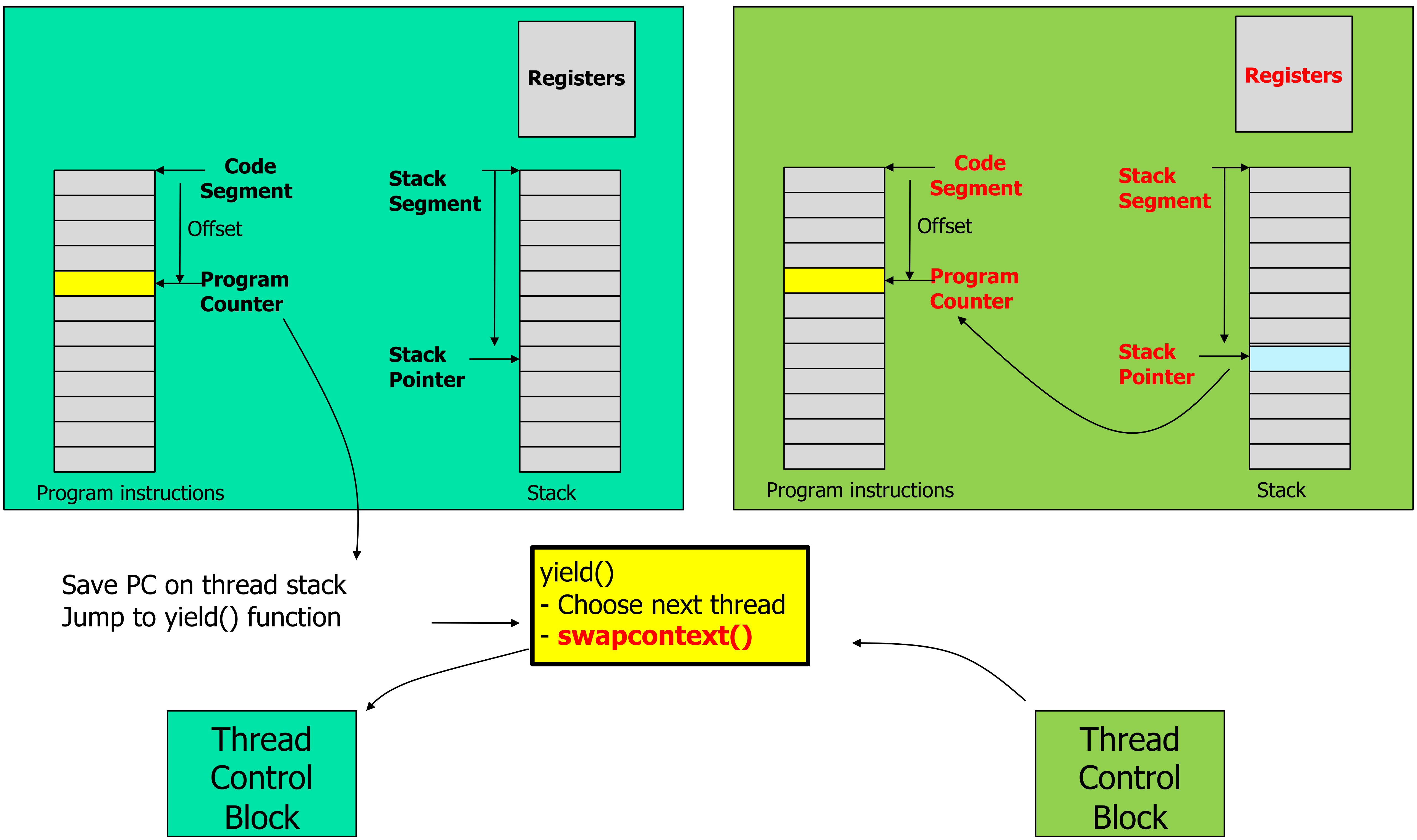
CTX Switch: Yield



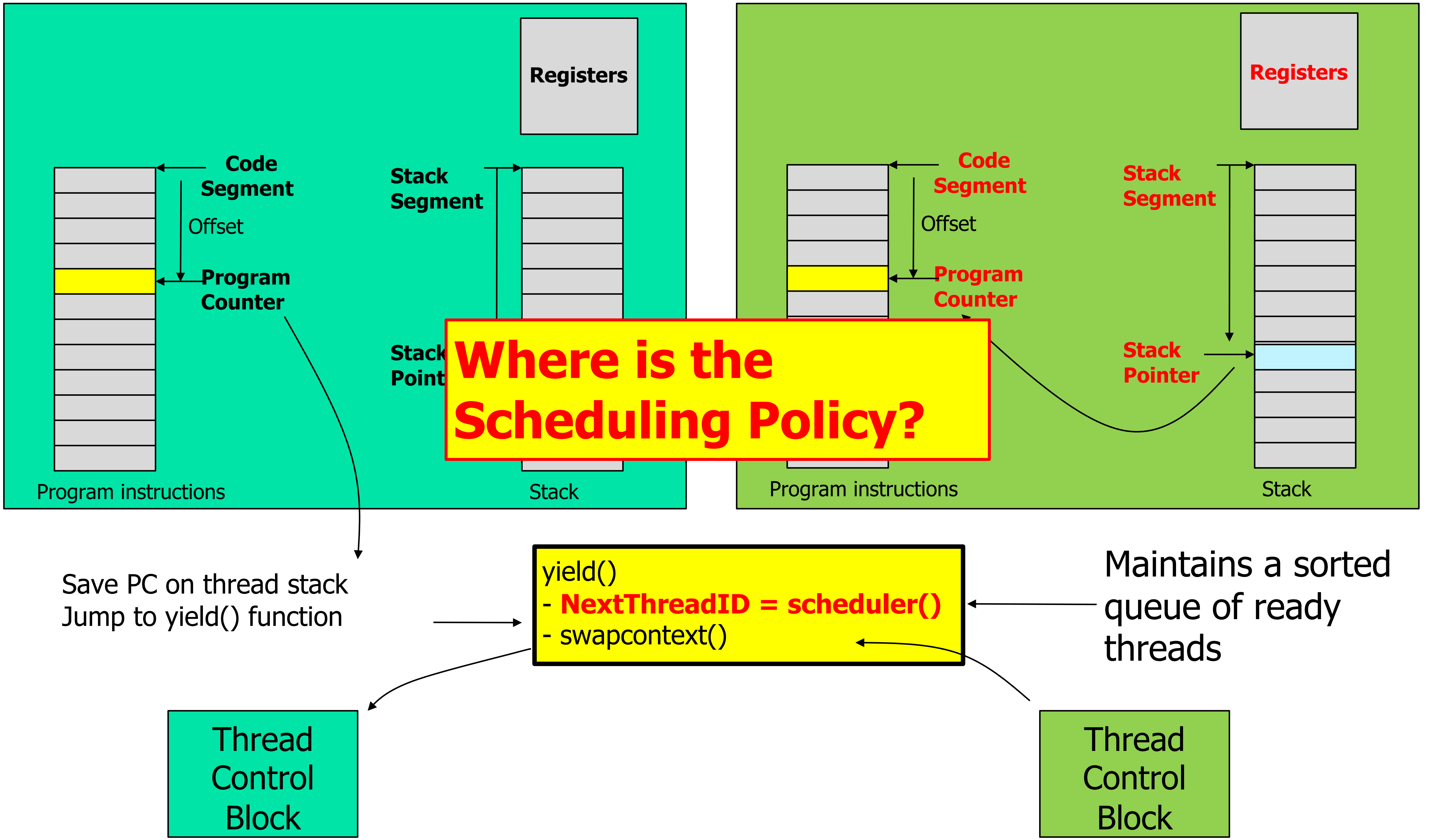
CTX Switch: Yield



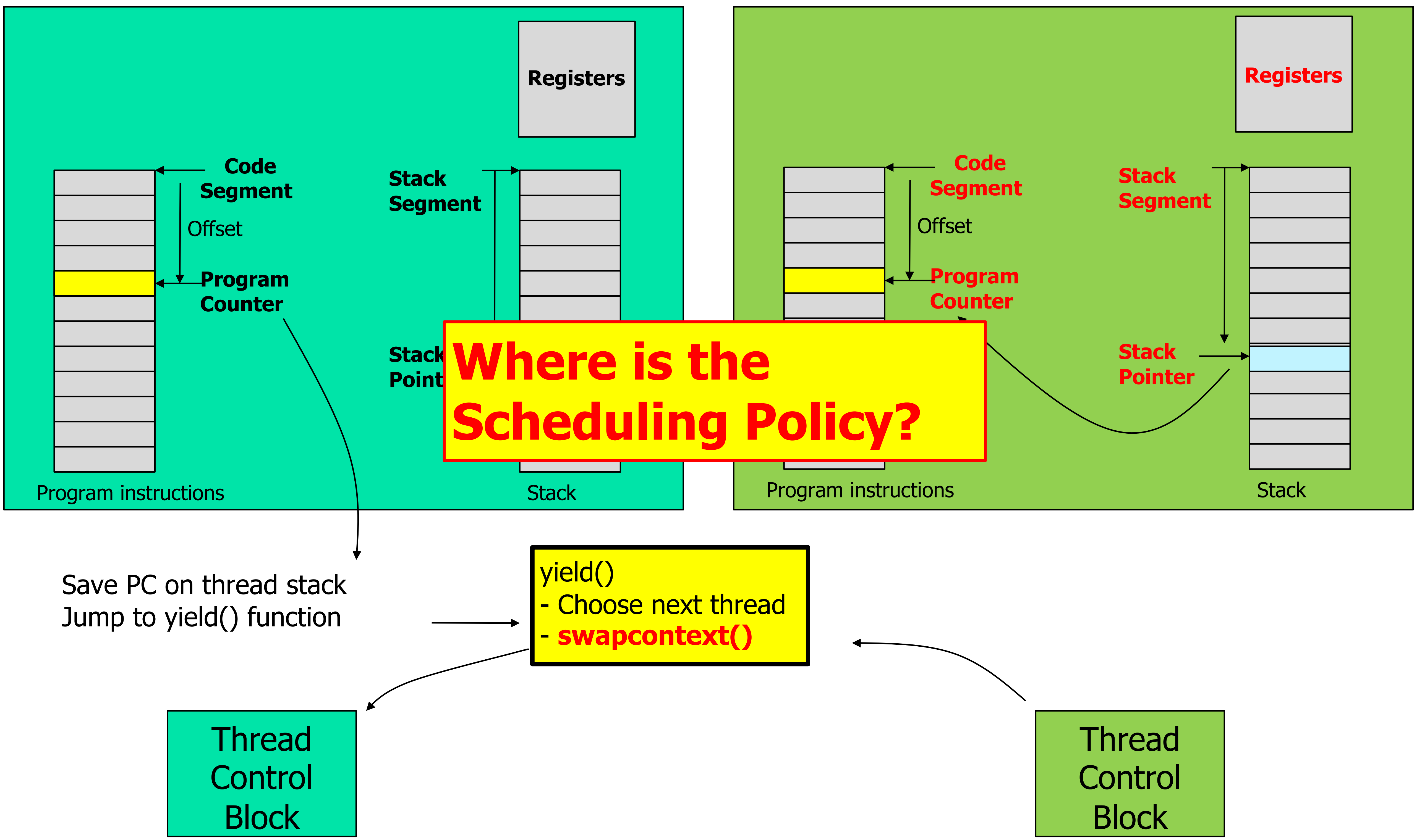
CTX Switch: Yield



Scheduler


Registers
Code Segment
Offset
Program Counter
Stack Segment
Stack Pointer
Program instructions
Stack
Registers
Code Segment
Offset
Program Counter
Stack Segment
Stack Pointer
Program instructions
Stack
Where is the Scheduling Policy?
Save PC on thread stack
Jump to yield() function
yield()
- NextThreadID = scheduler()
- swapcontext()
Maintains a sorted queue of ready threads
Thread Control Block
Thread Control Block

Scheduler



Registers
Code Segment
Offset
Program Counter
Stack Segment
Stack Pointer
Program instructions
Stack
Registers
Code Segment
Offset
Program Counter
Stack Segment
Stack Pointer
Program instructions
Stack
Where is the Scheduling Policy?
Save PC on thread stack
Jump to yield() function
yield()
- Choose next thread
- swapcontext()
Thread Control Block
Thread Control Block

Syscall

Must save callee registers and instruction pointer to resume after syscall

Where are these saved?

Kernel stack: every process has its own kernel stack

Operating System	Hardware	Program Process A
		Run main() ... Call system call trap into OS
	save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Do work of syscall return-from-trap		
	Restore regs (from kstack) move to user mode jump to PC after trap	

Syscall

How does the hardware know where to jump (i.e., trap handler location) !?

Solution: trap table and system call table

64 tells that this is syscall (other numbers for other exceptional events)

6 tells that this is a sys_read

During boot time, OS “configures” the hardware to say where the trap handlers are located…

On trap, hardware simply jumps to this location

OS then knows this is a syscall, uses the syscall number to decide which particular syscall to invoke

To call SYS_read the instructions we used were

movl $6, %eax
int $64

To call SYS_exec what will be the instructions?

movl _____ %eax
int _____

```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_write   5
#define SYS_read    6
#define SYS_close   7
#define SYS_kill    8
#define SYS_exec    9
#define SYS_open   10
```

TIMER-BASED INTERRUPTS

Option 2: Timer-based Multi-tasking

Guarantee OS can obtain control periodically

Enter OS by enabling periodic alarm clock

- Hardware generates timer interrupt (CPU or separate chip)
- Example: Every 10ms
- Can user code turn off timer?

Operating System | Hardware | Program Process A

Operating System	Hardware	Program
		Process A
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	

Operating System

Hardware

Program

Process A

timer interrupt
save regs(A) to k-stack(A)
move to kernel mode
jump to trap handler

Handle the trap
Call switch() routine
 save kernel regs(A) to proc-struct(A)
 restore kernel regs(B) from proc-struct(B)
 switch to k-stack(B)
 return-from-trap (into B)

Note: difference b/w caller vs. callee-saved registers

Operating System

Hardware

Program

Process A

timer interrupt
save regs(A) to k-stack(A)
move to kernel mode
jump to trap handler

Handle the trap
Call switch() routine
 save kernel regs(A) to proc-struct(A)
 restore kernel regs(B) from proc-struct(B)
 switch to k-stack(B)
 return-from-trap (into B)

restore regs(B) from k-stack(B)
move to user mode
jump to B's IP

Operating System	Hardware	Program
		Process A
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call switch() routine save kernel regs(A) to proc-struct(A) restore kernel regs(B) from proc-struct(B) switch to k-stack(B) —> this is the key point return-from-trap (into B)		
	restore regs(B) from k-stack(B) move to user mode jump to B's IP	
		Process B

xv6 Example

```c
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

- Scheduler switches to user process in “scheduler” function
- User process switches to scheduler thread in the “sched” function

```c
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&ptable.lock);

  }
}
```

Swtch() function

- What is on the k-stack of A when a process A has just invoked the swtch?
- What does swtch do?
- What will swtch find on new kernel stack?
- Where does it return to?

Swtch() function

•What is on the k-stack when a process A has just invoked the swtch?

Just the caller save registers of A, return address (eip) – where is this exactly?

• What does swtch do?

Push remaining registers on old kernel stack (i.e., callee save registers or kernel registers of A)

Save pointer to this context into context structure pointer of old process (A)

Switch esp from old kernel stack (A k-stack) to new kernel stack (B k-stack)

ESP now points to saved context of new process (B k-stack)

Pop callee-save registers from new stack

Return (pops return address, caller save registers – hardware does this)

• What will swtch find on new kernel stack?

Whatever was pushed when the new process gave up its CPU in the past

•Where does it return to?

We switched kernel stacks from old process to new process, CPU is now executing new process code, resuming where the process gave up its CPU by calling swtch in the past

Swtch impl in xv6

```
# void swtch(struct context *old, struct context *new);
#
# Save current register context in old
# and then load register context from new.
.globl swtch
swtch:
  # Save old registers
  movl 4(%esp), %eax   # put old ptr into eax
  popl 0(%eax)         # save the old IP
  movl %esp, 4(%eax)   # and stack
  movl %ebx, 8(%eax)   # and other registers
  movl %ecx, 12(%eax)
  movl %edx, 16(%eax)
  movl %esi, 20(%eax)
  movl %edi, 24(%eax)
  movl %ebp, 28(%eax)

  # Load new registers
  movl 4(%esp), %eax   # put new ptr into eax
  movl 28(%eax), %ebp # restore other registers
  movl 24(%eax), %edi
  movl 20(%eax), %esi
  movl 16(%eax), %edx
  movl 12(%eax), %ecx
  movl 8(%eax), %ebx
  movl 4(%eax), %esp   # stack is switched here
  pushl 0(%eax)        # return addr put in place
  ret                  # finally return into new ctxt
```