Operating Systems

Interrupts, Syscalls and Context Switch

Giorgio Grisetti
grisetti@diag.uniroma1.it

Department of Computer Control and Management Engineering
Sapienza University of Rome

Interrupts in OS

Modern Operating Systems are interrupt based

- Each interaction with the OS is triggered by an interrupt

How can an interrupt arise?

- External events (e.g. I/O, timer)
- Internal Exceptions (e.g. illegal instruction...)
- Explicit call(e.g. syscall)

Interrupts Why

Polling Option

- continuously query the status

```
while(1){
   if(key_pressed)
      handleKey();
   if(disk_finished)
      handleDisk();
      ....
}
```

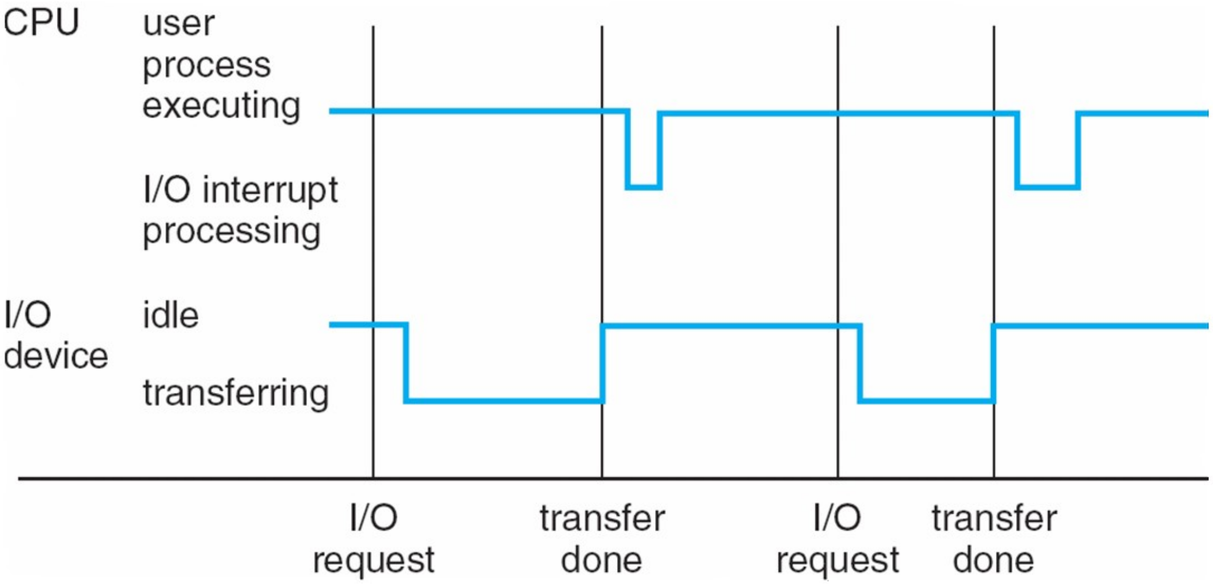
Interrupt option

- get woken up when something happens, but sleep most of the time

```
void keyISR();
void diskISR();
...
while(1){
   // Minimal Power
   halt();
}
```

Interrupt in OS

- When an interrupt occurs, the current context is saved, and the CPU starts executing the interrupt identification routine
- Usually
  - a single physical interrupt manages multiple devices (e.g. USB), thus figuring out who was responsible of the interrupt requires little handshaking
  - while serving an interrupt, interrupts of the same type are disabled
- **The ISR is part of the OS**



Interrupt Vector

- Is an array of function pointers containing the addresses of the ISR
- Each location is associated to a specific event

Interrupt ID	ISR pointer
0 (es. reset)	ADR 0
1 (es. serial)	ADR 1
2 (es. TRAP)	ADR 2
3 (...)	ADR 3
……...	……...

Exceptions

The exceptions are software triggered interrupts.

On x86, they fall in these categories:

- Traps: ISR is invoked after triggering instruction.

  examples: INT instruction, Breakpoint

- Faults: ISR is invoked before triggering instruction

  examples: divide by 0, page fault, illegal instruction

- Aborts: state of the triggering process cannot be recovered

  example: double fault exception

The CPU deals with its circuits to the occurrence of these events.

For each event there is an entry in the interrupt vector.

Dealing with these events is a task of the OS.

INT and CALL

Difference between explicilty called ISR and calling a soubroutine:

INT <XX>

- behavior: jump to ISR whose address is stored in position <XX> of interrupt vector
- <XX> is an index of the ISR vector (limited number)
- there is a **limited** number of **controlled** entry points for the INT instruction
- the cpu flags are altered when jumping to the ISR. (Supervisor Mode is entered)

CALL <YY>

- behavior: call soubroutine whose address is YY>.
- <YY> can be **any valid address** mapped in the executable memory area of a process.
- the flags are NOT altered

Dual Mode

Program misbehaving:

- What happens if a user program alters the interrupt vector?
- What happens when a user program writes random stuff on a memory mapped device (e.g. the disk controller)?

  <your answers>

The OS needs to have control of int vector and I/O ports to do his job.

Solution:

- prevent the program to do this by defining two operation modes for the CPU: **privileged** and **user** mode.
- in user mode only a subset of the instructions can be executed
- the modes are toggled by a bit in the FLAG register
- altering this flag is a privileged instruction
- ISR are always executed in **privileged** mode

Dual Mode

OS is executed in privileged mode, user program not.

Issue:

Calling the OS from user program requires changing the flags, but this is a priviledged instruction.

Solution:

- hide the entire OS behind an entry of the interrupt vector.
- the specific OS function is invoked through an INT <OS_ISR> instruction (syscall).
- parameters of the syscall can be:
  - in the CPU registers
  - on the stack

Issue:

- what if the used program does "INT <DISK_ISR>"?

Solution:

- only a subset of the location in the interrupt vector can be called by the user program without generating a protection exception.

Syscall

One single controlled entry point to the OS nucleum (kernel)

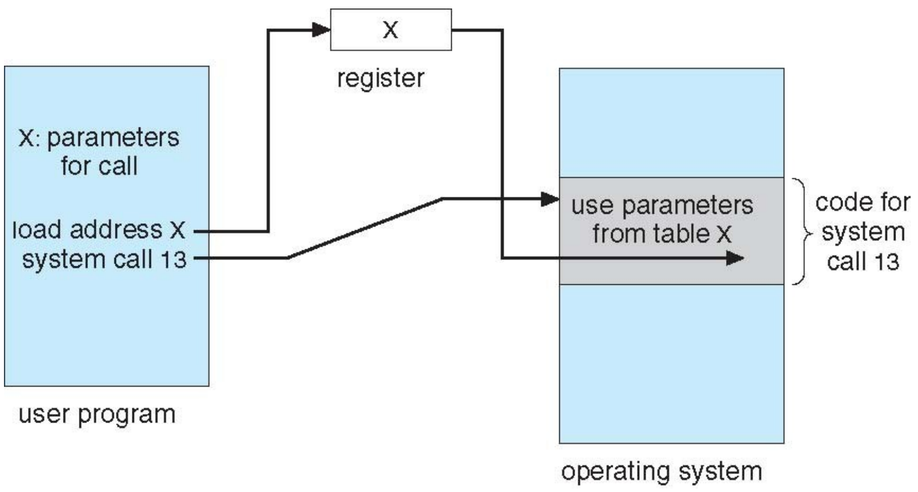
User programs are "caged" :)

Problem:

The kernel may offer several functionalities, but we have one single ISR to handle all of them

Solution:

- enumerate all possible functions (syscall number)
- use a register to select which function to invoke (EAX on Linux-x86)
- the specific syscall looks up the remaining parameters on the register/stack

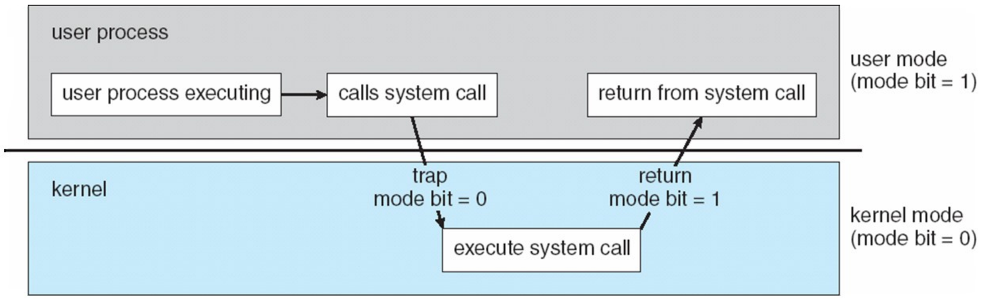


Problem:

How to restore the mode-bit after executing a syscall?

Solution:

Done by the IRET (return from interrupt), instruction that restores the flags



Typical Syscalls

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information

Portability

The syscalls usually offer rather low level functionalities

- write/read n bytes on a device
- map a certain amount of RAM in the memory space of a process
- open/close a device
- wait for some data to become available
- ....

Albeit these functionalities typically enable the development of complex applications, operating at system call level would result in a tight dependancy on the OS

The software should be rewritten for each OS

Standards

Language Standards

- Some languages come with a standard library, that offer I/O functionalities through high level functions
  - (f)printf, fopen, fclose

  Writing programs using only functions in the standard library ensures portability
- The implementation of these functions rolls back to specific syscalls

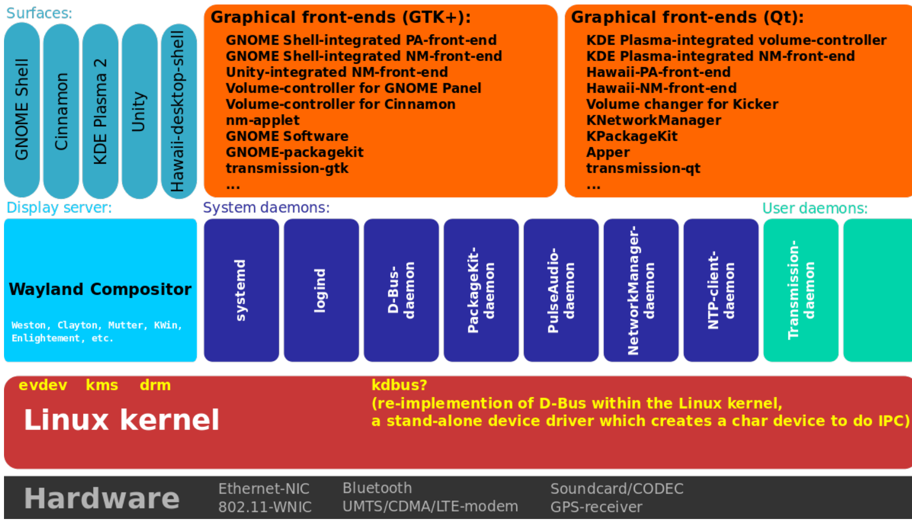
System Standards

- More powerful functionalities of the os are usually not covered by the standard library of the language.
- To ease portability among different OSes, some committee has defined standard libraries that provide these functionalities
  - threads, network, synchronization
- Examples: POSIX (Linux, Solaris, Darwin) vs WinAPI (Windows)

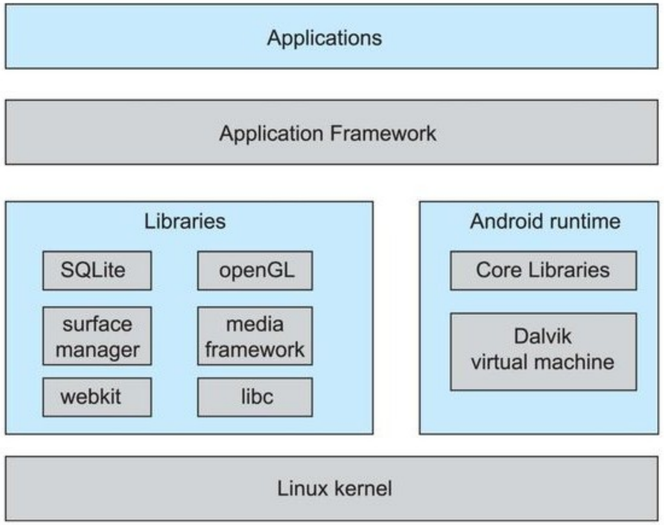
Layers

- A core rule in the design of the sistem is to define a layered architecture.
- The functionalities at higher layer are built exclusively on top of functionalities at the lower layer.
- When desiging an application, always use the highest possible layer to ensure broader portability.

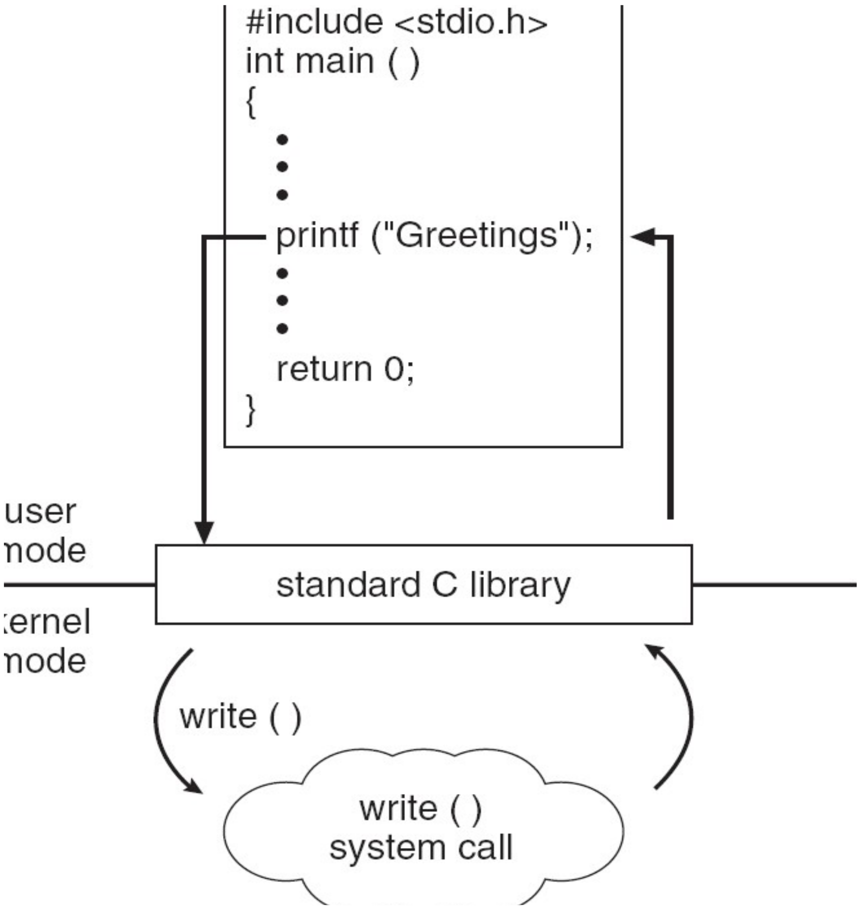
linux desktop

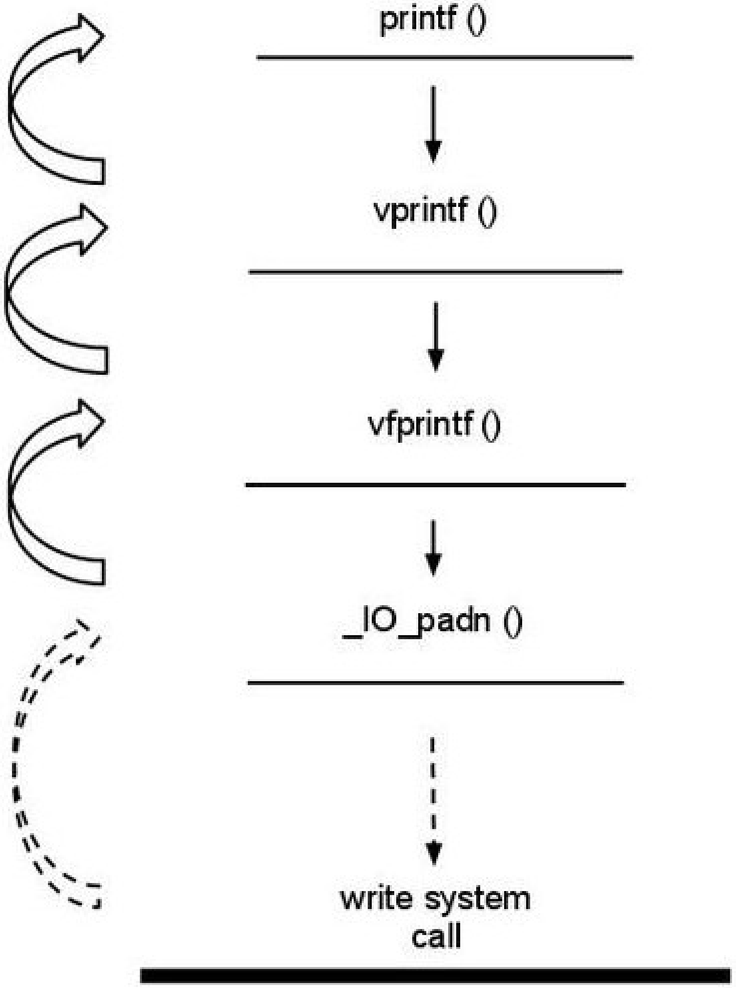


android



Example: printf





Process Control Block

The kernel stores the information about a process in a data structure: the Process Control Block (PCB)

A typical PCB contains

- process ID (PID)
- user ID (UID)
- status of the program (ready, waiting...)
- CPU status for the process (registers)
- scheduling information*
- memory information (stack, page table*)
- I/O information (open descriptors*)

All in all from the PCB and the data structures linked by it one should be able to recover the state of a process.

The PCB is in a privileged memory area.

* on this screen, in the next episodes

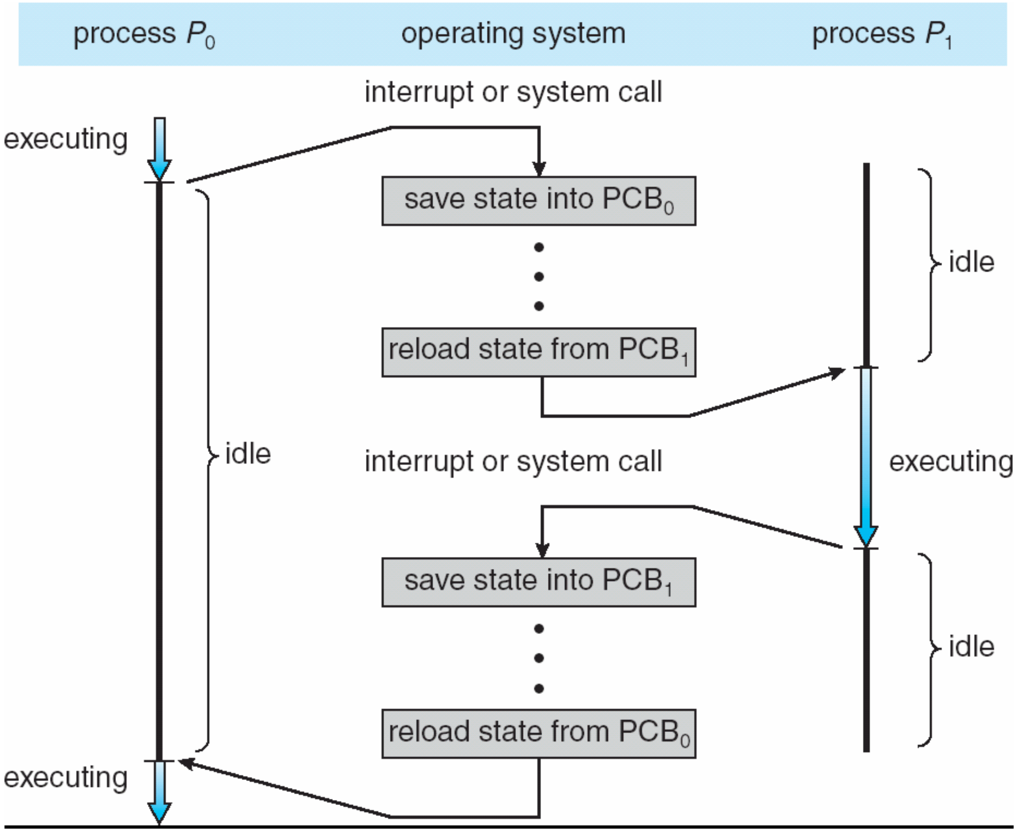
Context Switch

Occurs when a running process is interrupted.

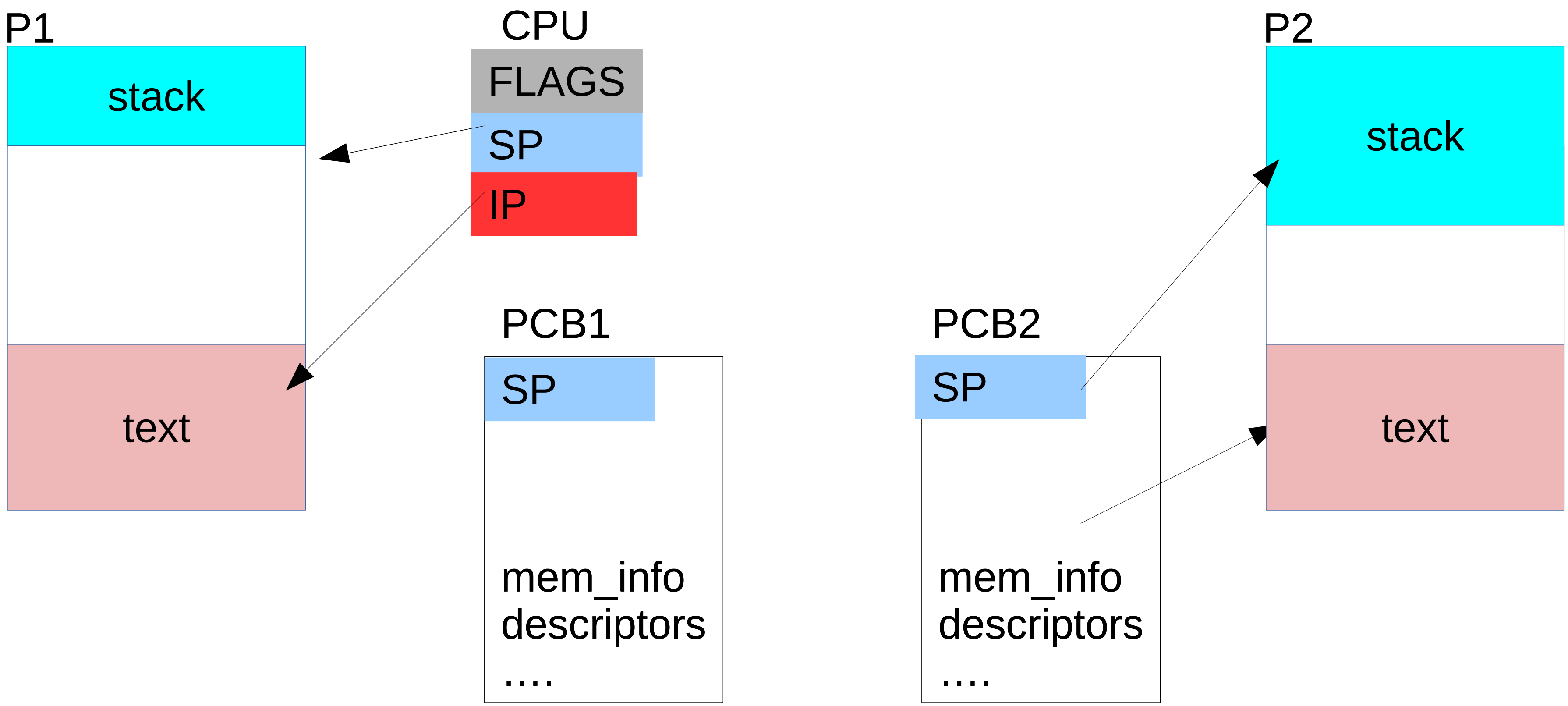
This can happen only because of an interrupt (or exception*).

These events cause the execution of kernel code.

When the kernel code returns, the next process that will enter the CPU might or might not be the one interrupted, depending on kernel's decision.



Context Switch in Detail

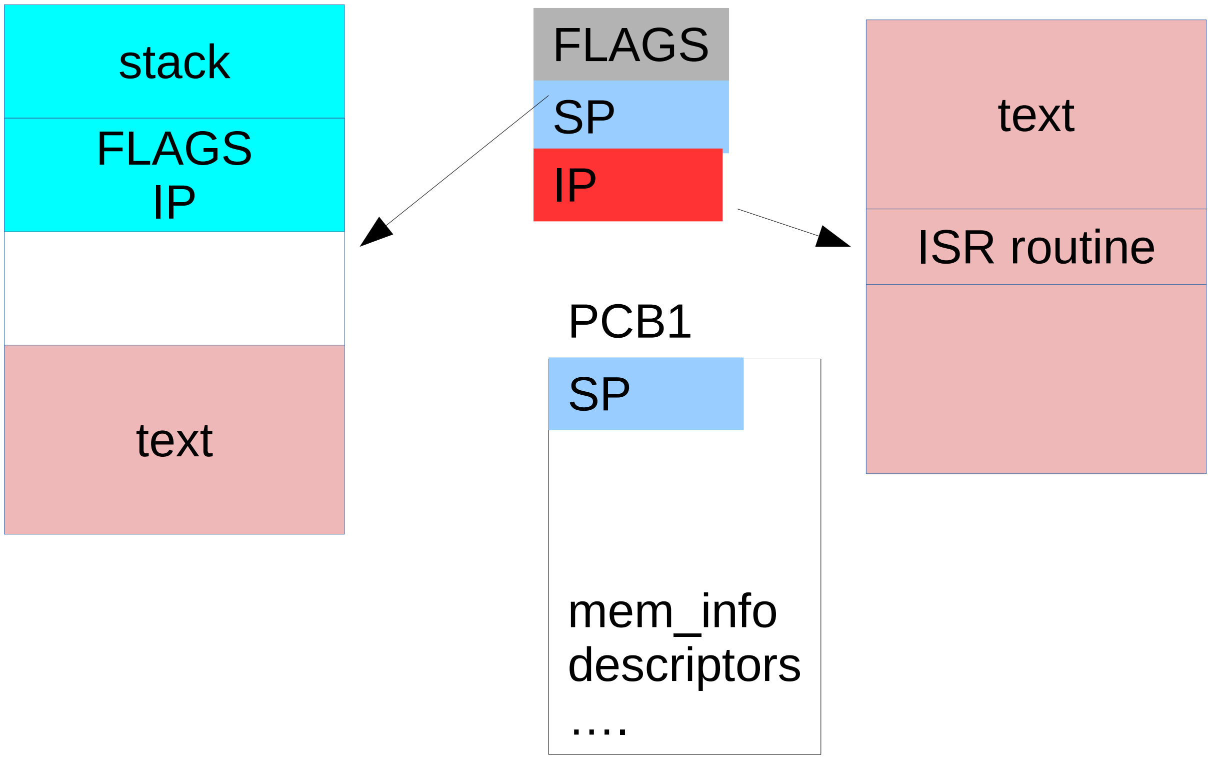


Scenario

- we have two processes in ready: P1 and P2
- P2 was previously sunning but it has been preempter. His status is in PCB2
- P1 is running

What should happen such that after an interrupt the CPU continues executing P2?

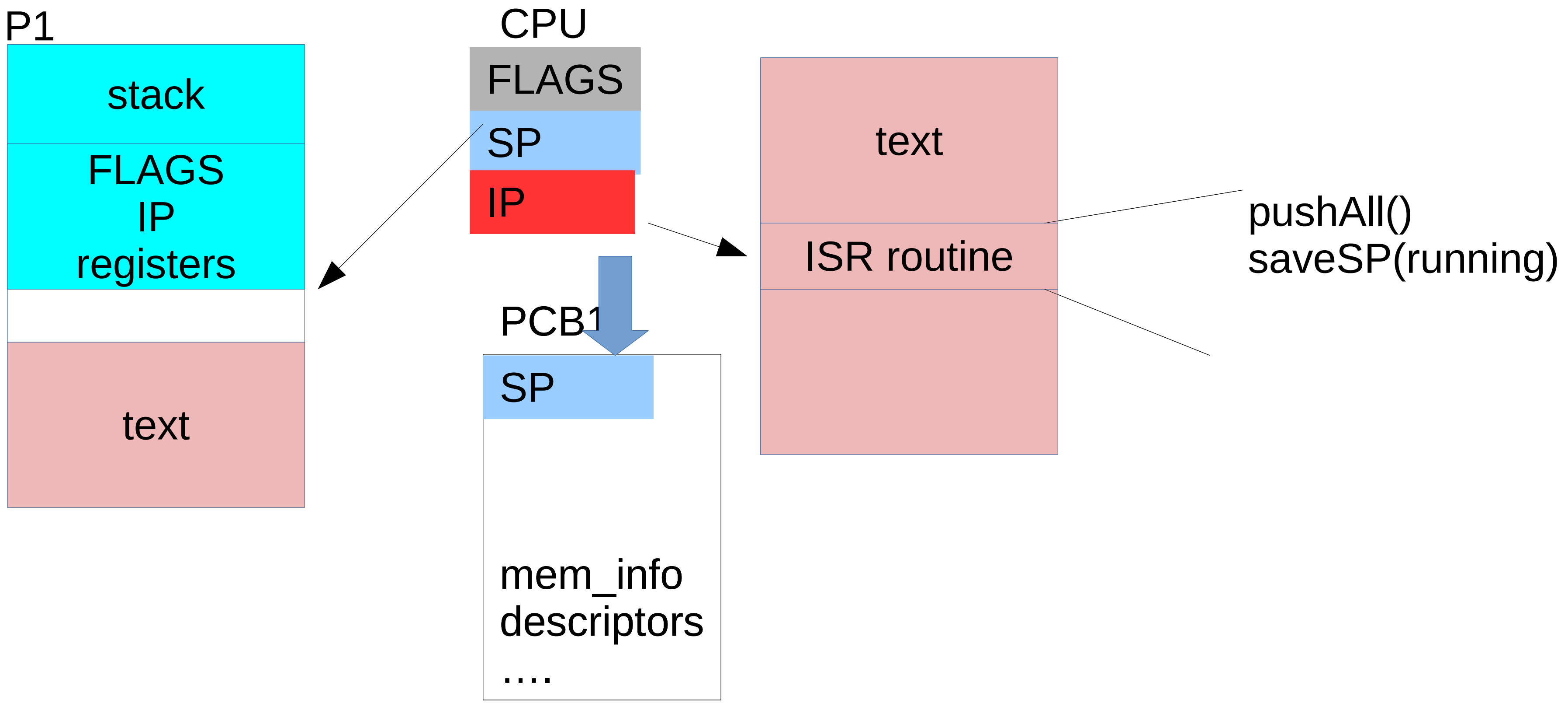
Context Switch in Detail



The interrupt comes, thus the CPU

- saves flags and instruction counter on the stack
- toggles to priviledged mode and handles the appropriate ISR

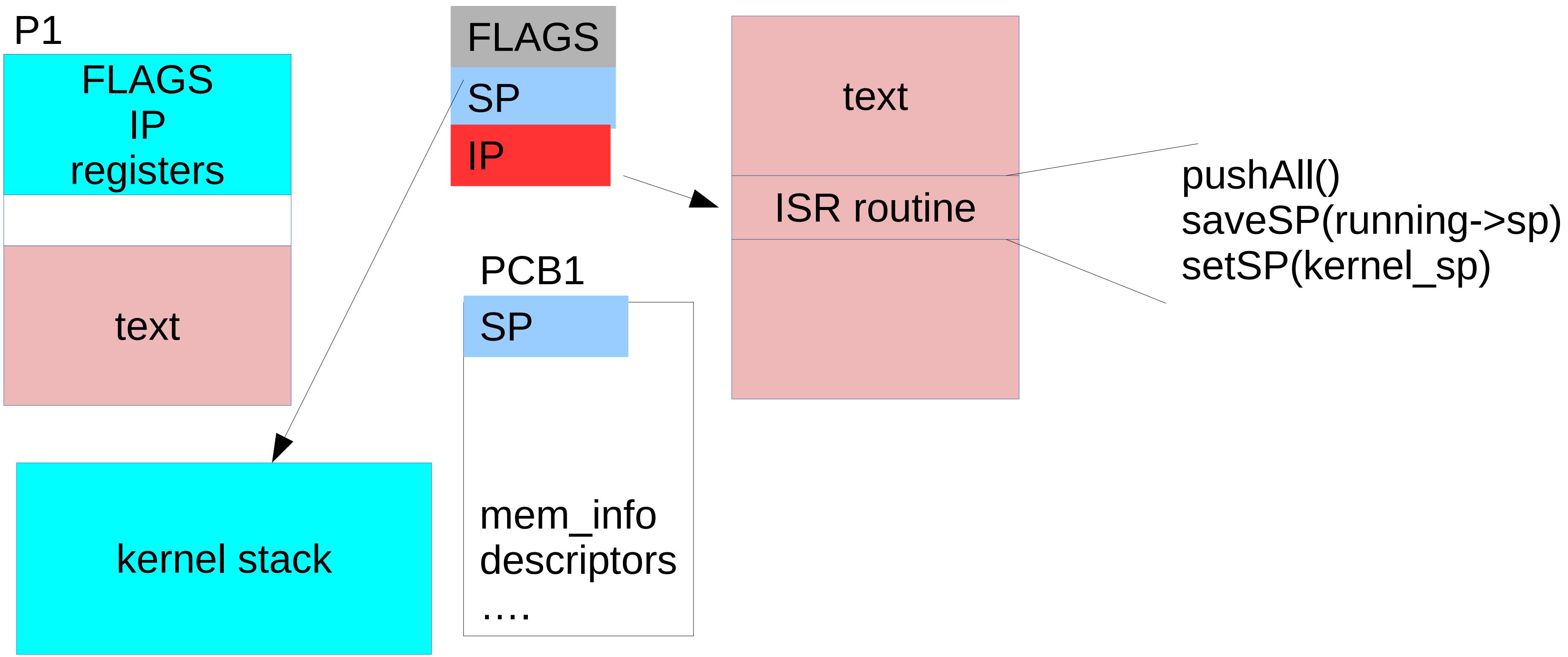
Context Switch in Detail



To recover P1 in the future, we need to save its CPU state in the PCB.

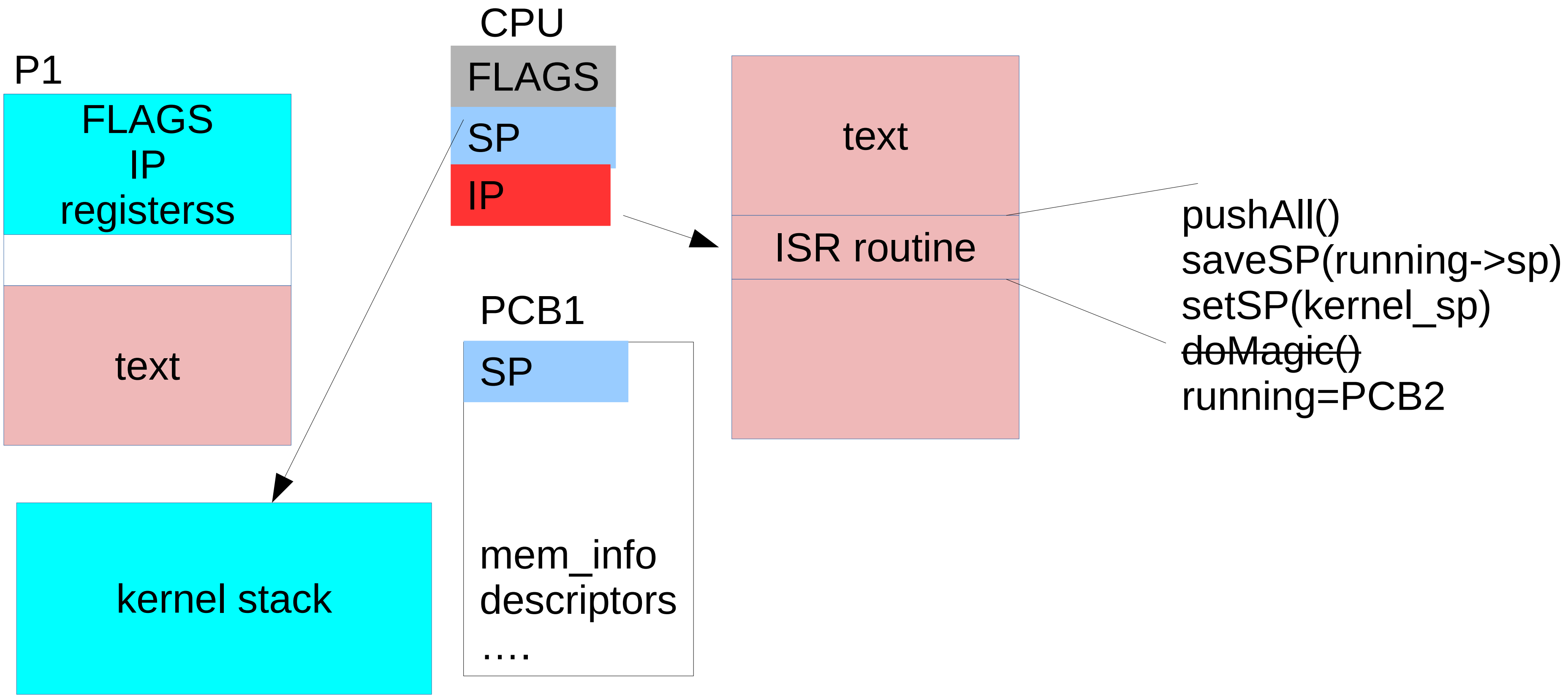
The state is on the stack, so in this example we save in the PCB just the stack pointer.

Context Switch in Detail



At this point we switch stack to the kernel stack. This step is optional, but it protects us from messing with the P1 stack.

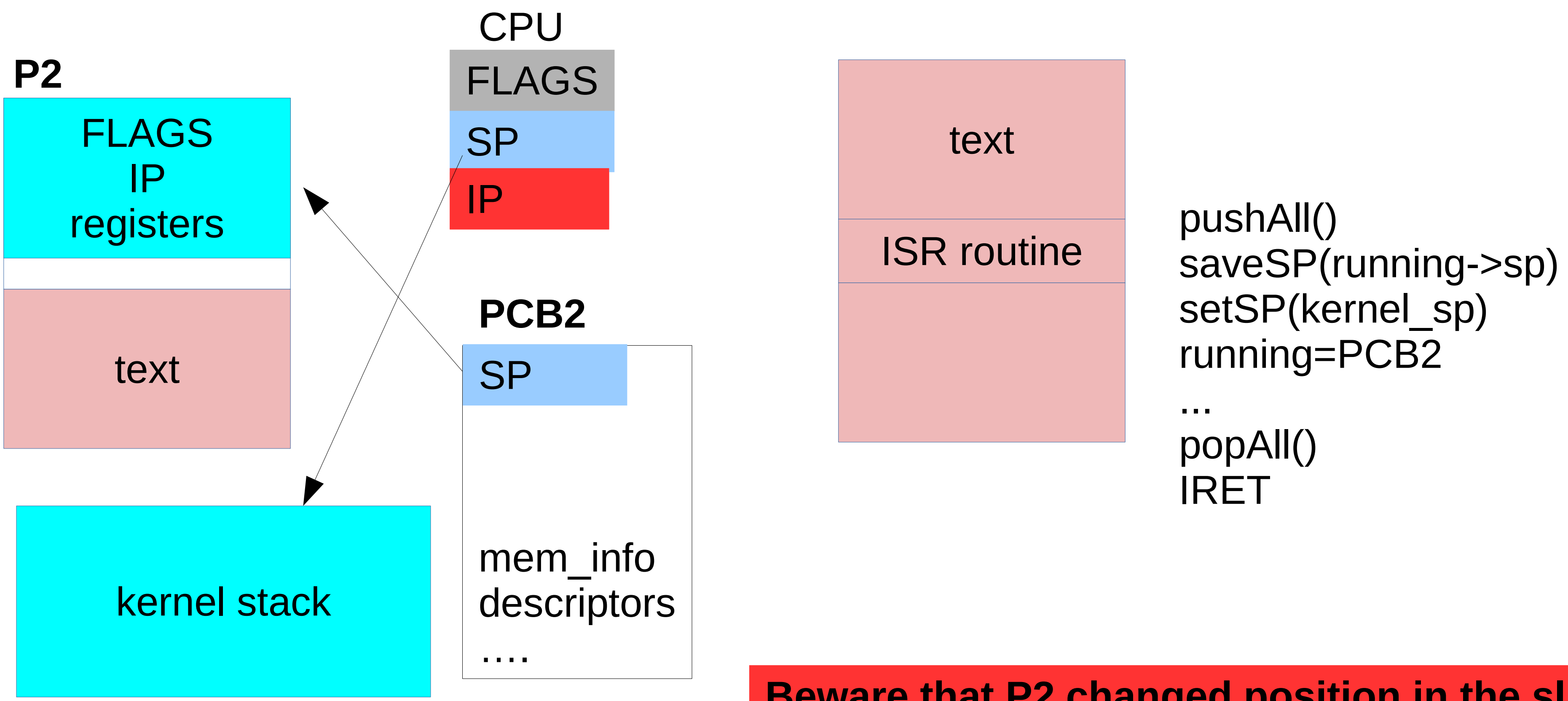
Context Switch in Detail



Once we change stack we do our task.

- If the trap was triggered by a syscall, we might want to look up for the parameters on the PCB or on P1's stack
- If our task is just to execute a context switch to P2, we need to select P2 as next running

Context Switch in Detail



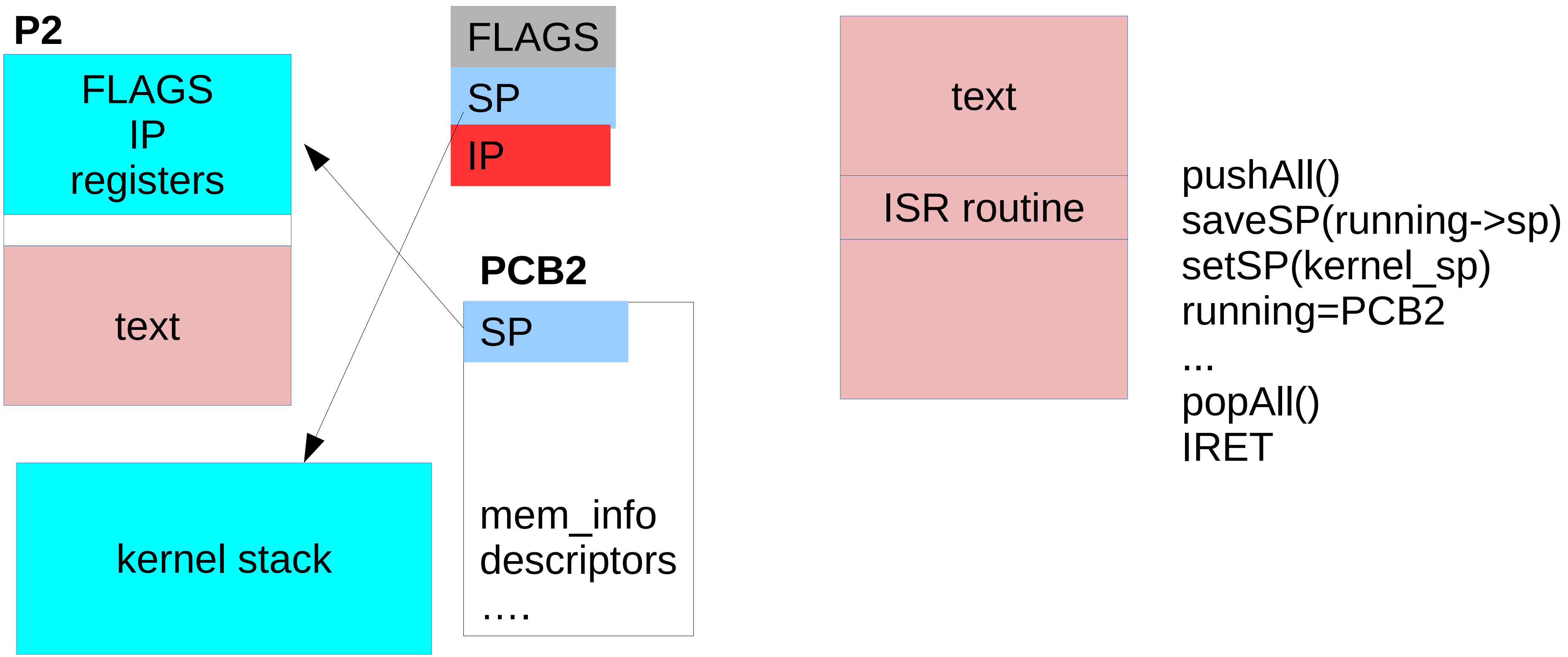
Beware that P2 changed position in the slide

Let us assume P2 is our next running, we need to start it again.

Since P2 was preempted, we know its structures are consistent

We know that the last instruction being executed in kernel mode will be a return from interrupt (IRET), that recovers the flags.

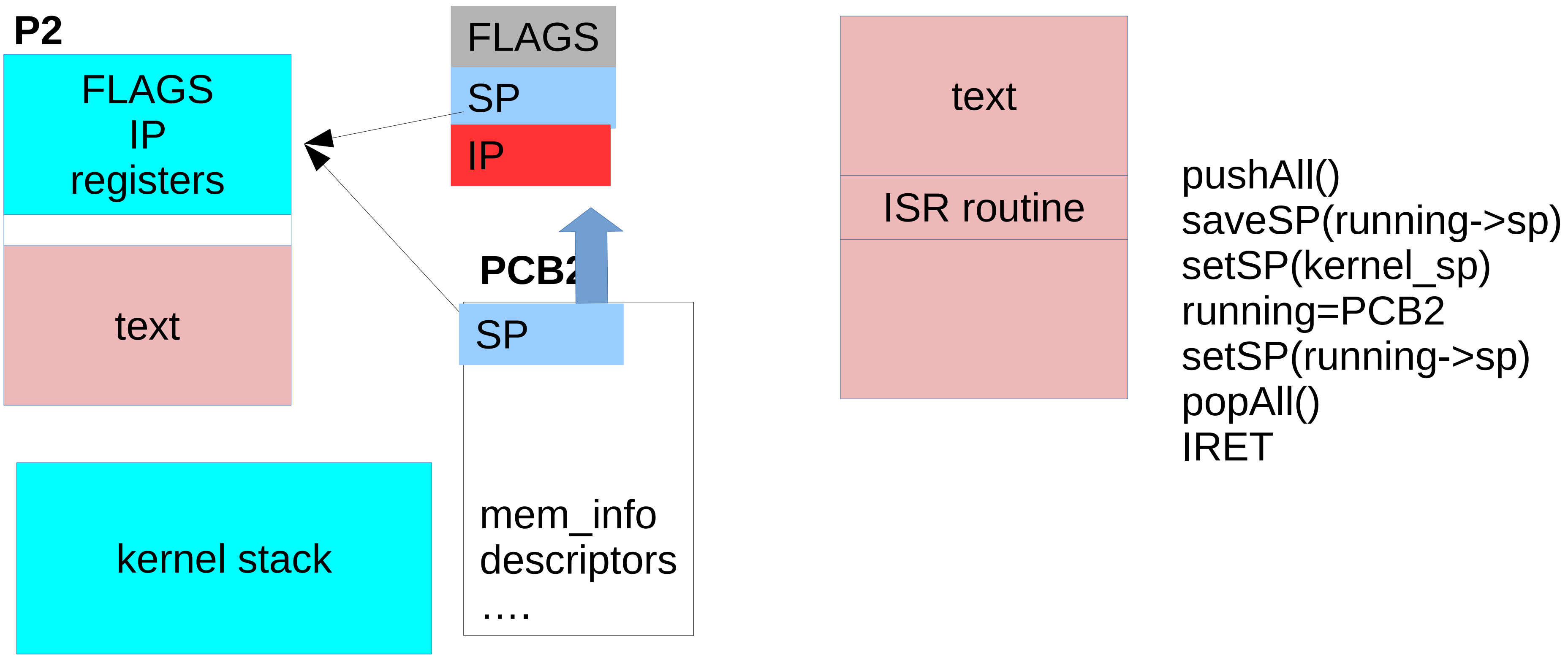
Context Switch in Detail



For the IRET to work, we need to assume the stack consistent.

This is verified since P2 was assumed to be preempted thus we "left" the stack untouched, after saving the registers

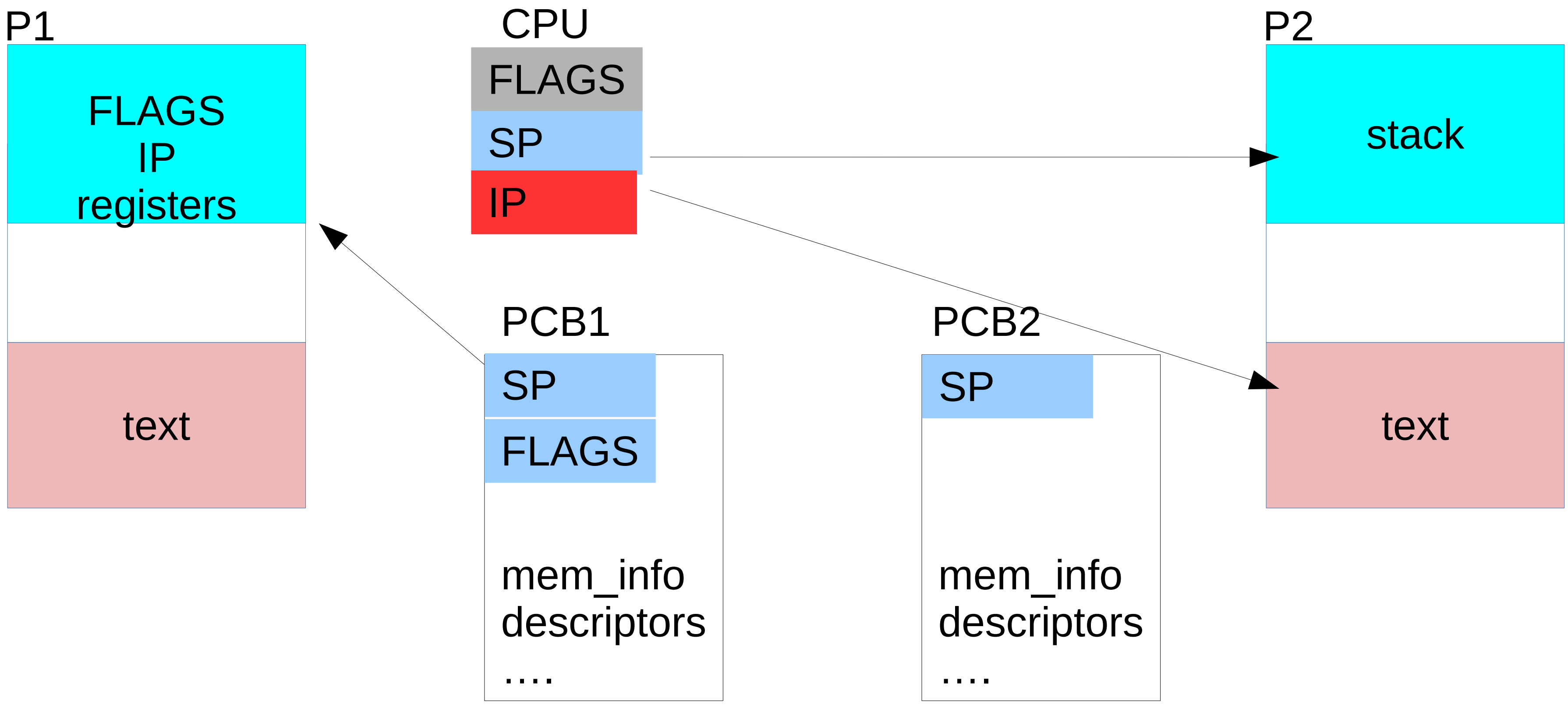
Context Switch in Detail



To continue the execution, we

- change the stack back resding it from the running pcb
- restore the state in the CPU
- return from interrupt

Context Switch in Detail



Et voila' P2 is running again as if nothing has happened

Preamble and Postamble

```
pushAll()
saveSP(running->sp)
setSP(kernel_sp)
```

```
doMagic()
```

```
setSP(running->sp)
popAll()
IRET
```

A generic ISR does not follow usual C calling conventions.

- The entry/exit in kernel mode is has a preamble and postamble, and have the role of ensuring a proper restoring of the process, and interaction with the kernel structures.
- Assembly needed for manipulating registers (SP, push).
- If a syscall wants to read some argument, it retrieves them from the stack of the current process (or from the registers), accessible through the SP saved in the current pcb.
- returning values done by altering the stack of the current PCB, in the area of the saved registers.