Sistemas Operativos 72.11

TP2: context switch, scheduling y procesos

Repasemos

¿Qué forma parte del contexto de un proceso?

¿Qué hacemos con esta información cuando no está ejecutando?

¿Cómo funcionan **push** y **pop**?

Description

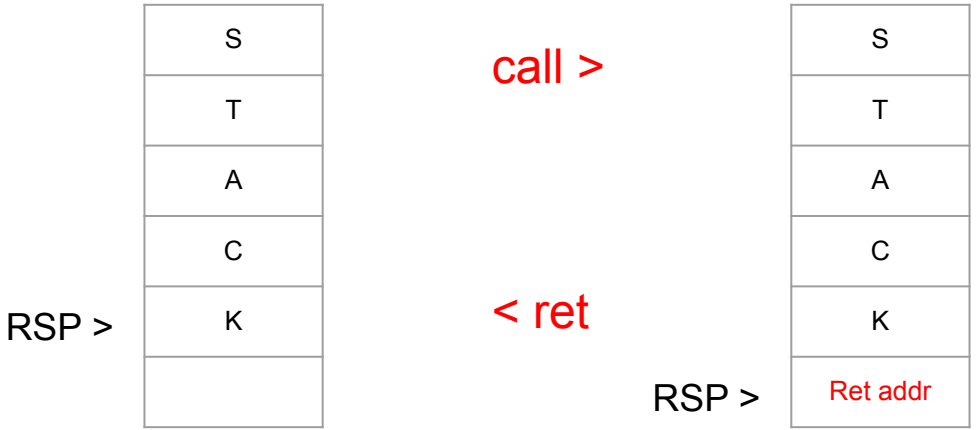
Decrements the stack pointer and then stores the source operand on the top of the stack. Address and operand sizes are determined and used as follows:

Description

Loads the value from the top of the stack to the location specified with the destination operand (or explicit opcode) and then increments the stack pointer. The destination operand can be a general-purpose register, memory location, or segment register.

Repasemos

¿Cómo funcionan **call** y **ret**?



Description

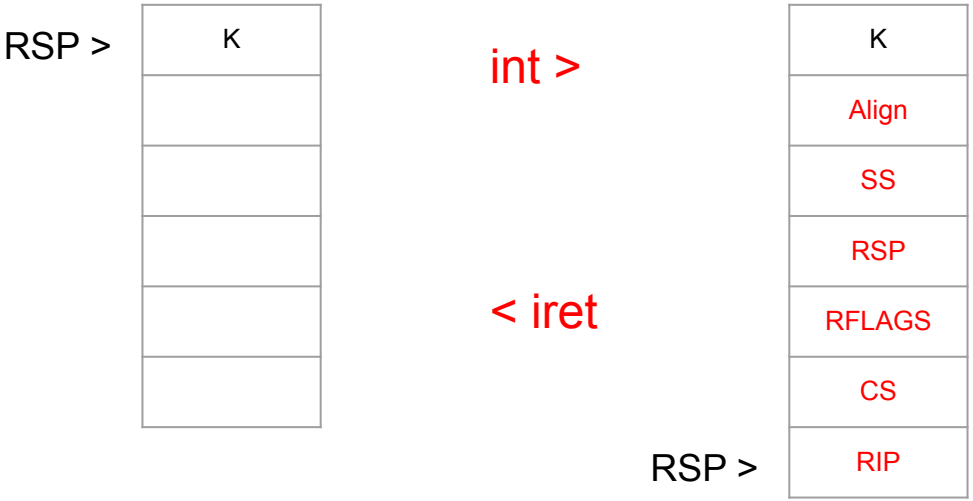
Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

Description

Saves procedure linking information on the stack and branches to the called procedure specified using the target operand. The target operand specifies the address of the first instruction in the called procedure. The operand can be an immediate value, a general-purpose register, or a memory location.

Repasemos

¿Cómo funcionan una **interrupción** y **iret**?



6.12 EXCEPTION AND INTERRUPT HANDLING

The processor handles calls to exception- and interrupt-handlers similar to the way it handles calls with a CALL instruction to a procedure or a task. When responding to an exception or interrupt, the processor uses the exception or interrupt vector as an index to a descriptor in the IDT. If the index points to an interrupt gate or trap gate, the processor calls the exception or interrupt handler in a manner similar to a CALL to a call gate (see Section 5.8.2, "Gate Descriptors," through Section 5.8.6, "Returning from a Called Procedure"). If index points to a task gate, the processor executes a task switch to the exception- or interrupt-handler task in a manner similar to a CALL to a task gate (see Section 7.3, "Task Switching").

ret es a **call** lo que … es a …

Desarrollemos un handler (dummy)

```
1.  dummy_handler:
2.    ; Send EOI
3.    mov al, 20h
4.    out 20h, al
5.
6.    iretq
```

¿Qué hace este handler?

¿Qué pasa con el estado del proceso?

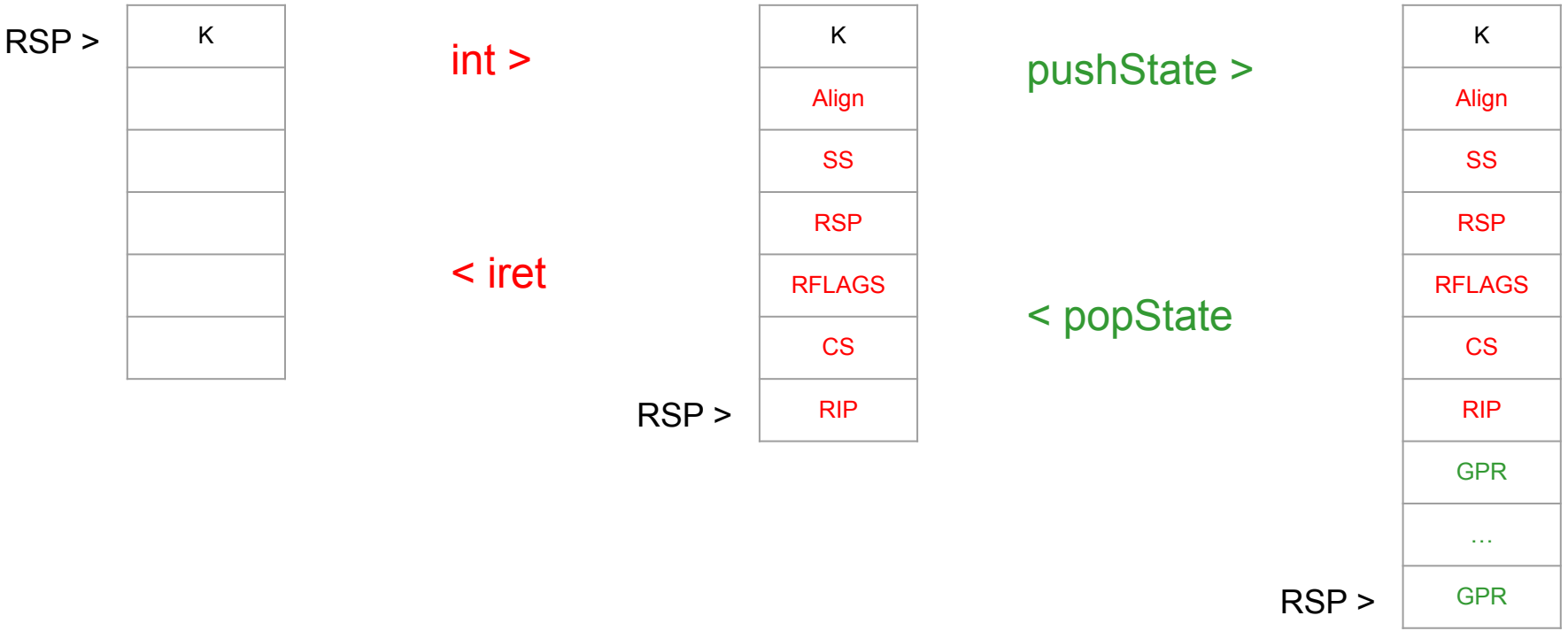
¿Dónde lo podemos guardar?

Desarrollemos un handler (still dummy)

```
1.  still_dummy_handler:
2.
3.    pushState
4.
5.    ; Send EOI
6.    mov al, 20h
7.    out 20h, al
8.
9.    popState
10.
11.   iretq
```

¿Cómo queda el stack justo después de pushState?

Desarrollemos un handler (still dummy)



GPR: Registros de propósito general apilados por pushState.

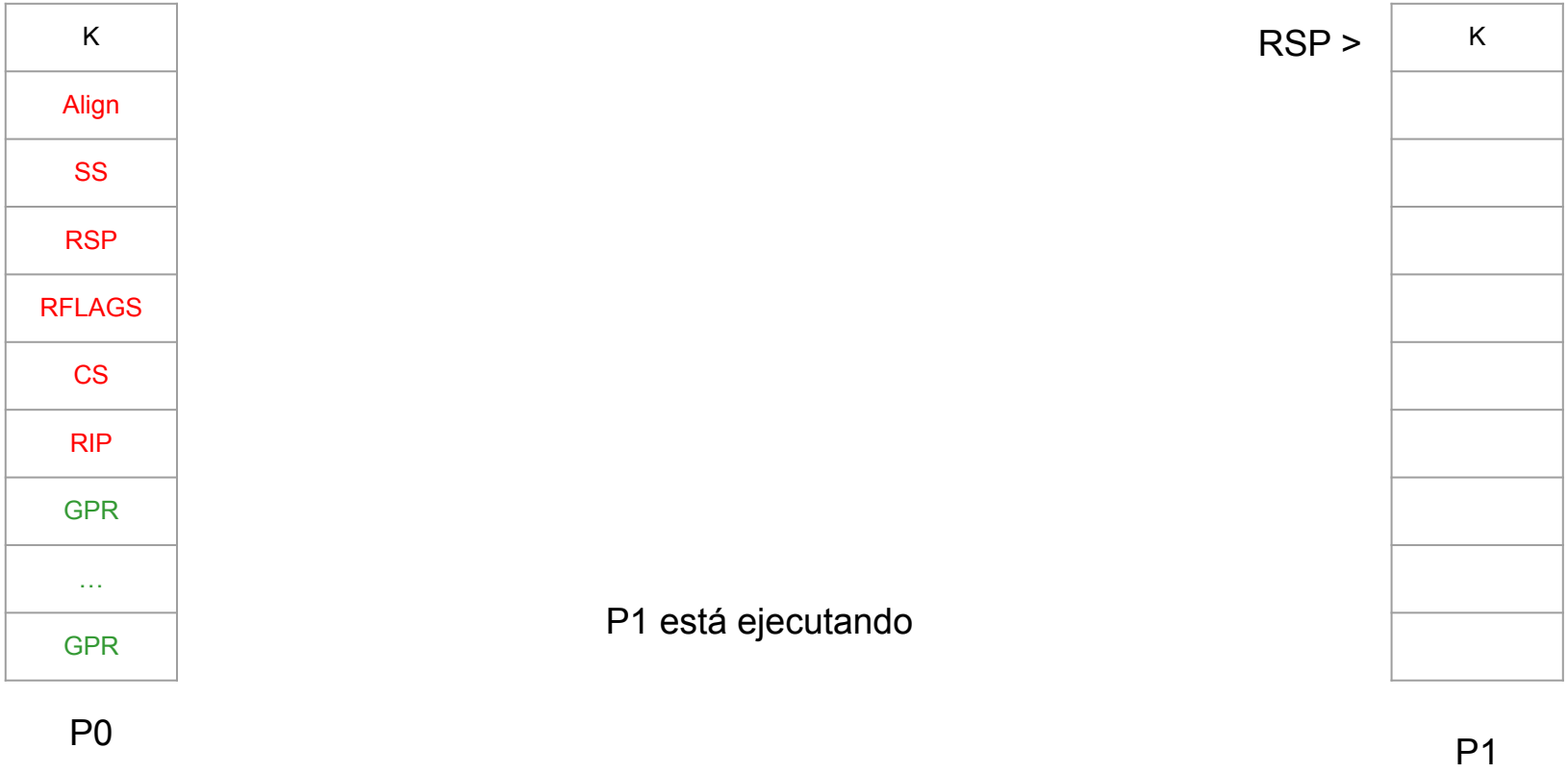
Desarrollemos un handler (still dummy)

```
1.  still_dummy_handler:
2.
3.    pushState
4.
5.    ; Send EOI
6.    mov al, 20h
7.    out 20h, al
8.
9.    popState
10.
11.   iretq
```

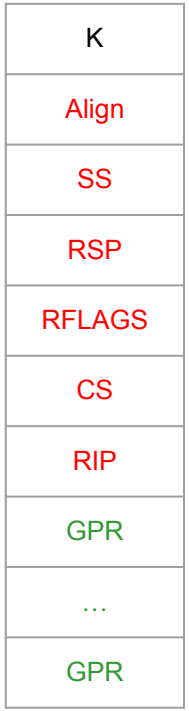
¿Por qué send EOI **luego** de pushState?

¿Qué hizo este handler?

Simulemos con 2 procesos P0 y P1



Simulemos con 2 procesos P0 y P1



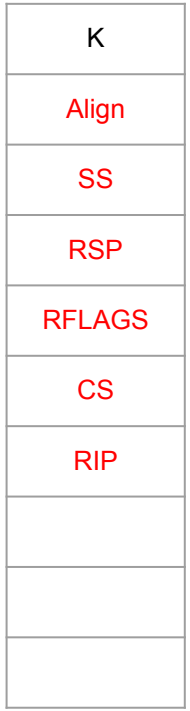
P0

RIP >

```
1.  still_dummy_handler:
2.
3.    pushState
4.
5.    ; Send EOI
6.    mov al, 20h
7.    out 20h, al
8.
9.    popState
10.
11.   iretq
```

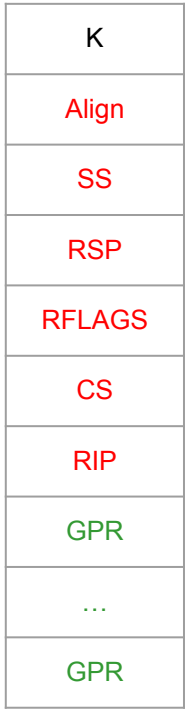
P1 es interrumpido por el timer

RSP >



P1

Simulemos con 2 procesos P0 y P1



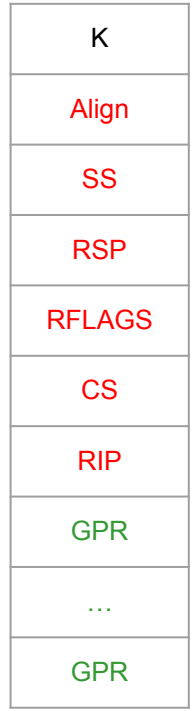
P0

RIP >

```
1.  still_dummy_handler:
2.
3.    pushState
4.
5.    ; Send EOI
6.    mov al, 20h
7.    out 20h, al
8.
9.    popState
10.
11.   iretq
```

P1 ejecuta pushState

RSP >



P1

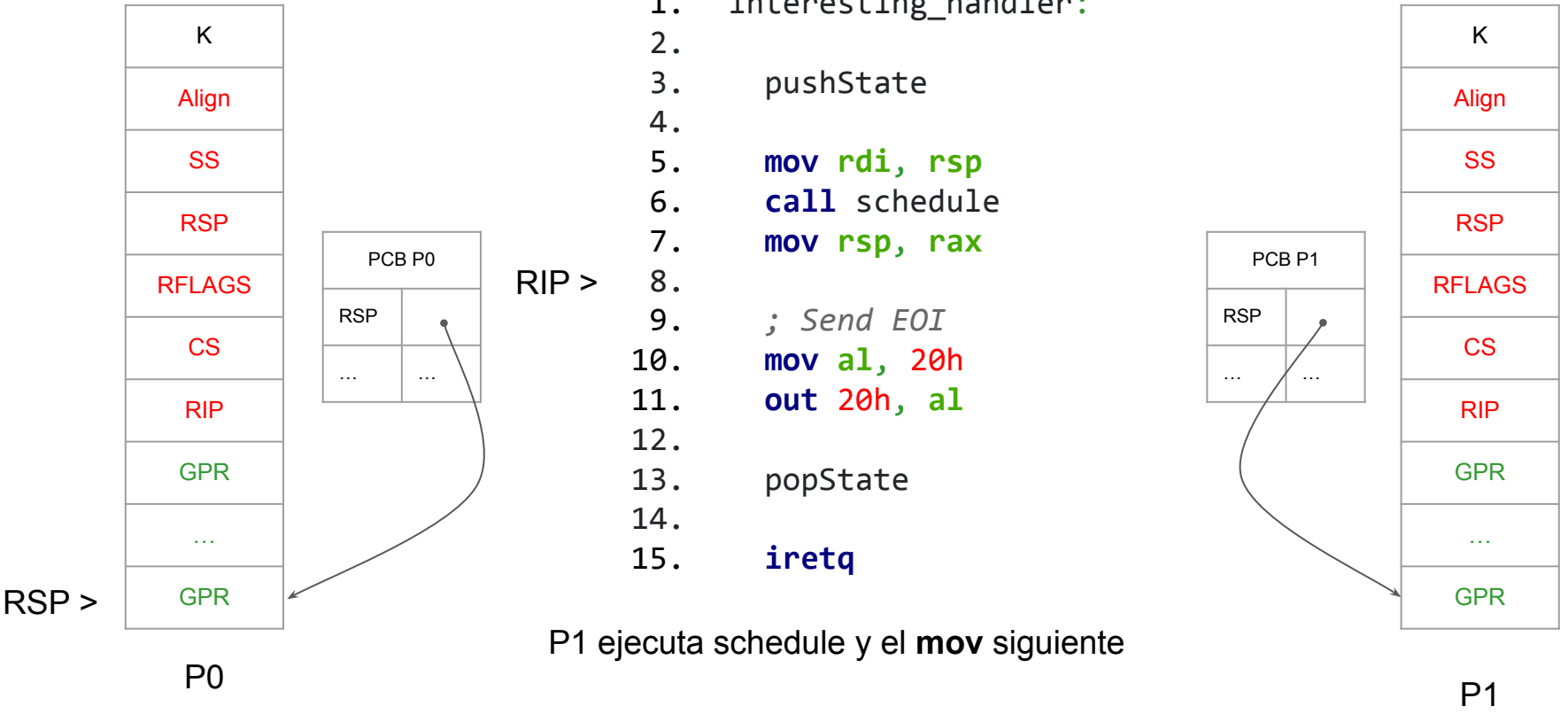
¿Qué debería ocurrir en este momento para que continúe ejecutando P0?

Simulemos con 2 procesos P0 y P1

Guardamos RSP en algún lugar (¿dónde?) y restauramos el RSP de P0 (¿desde dónde?)

```
1.  interesting_handler:
2.
3.    pushState
4.
5.    mov rdi, rsp
6.    call schedule
7.    mov rsp, rax
8.
9.    ; Send EOI
10.   mov al, 20h
11.   out 20h, al
12.
13.   popState
14.
15.   iretq
```

Simulemos con 2 procesos P0 y P1

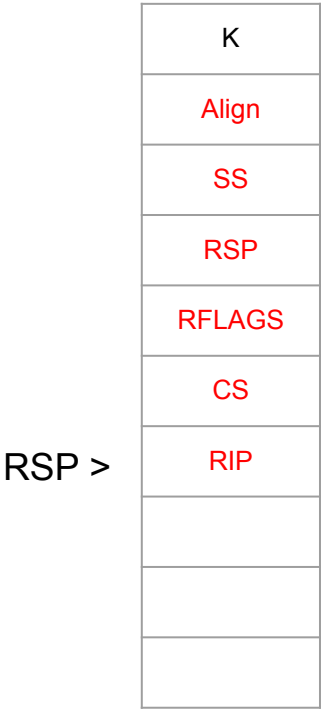


P1 ejecuta schedule y el **mov** siguiente

¿Dónde quedó el RIP? -> ¿Qué ocurre ahora?

ITBA
Instituto Tecnológico de Buenos Aires

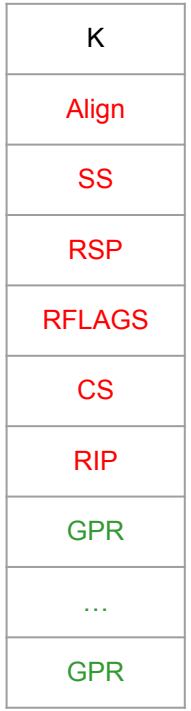
Simulemos con 2 procesos P0 y P1




```
1.  interesting_handler:
2.
3.    pushState
4.
5.    mov rdi, rsp
6.    call schedule
7.    mov rsp, rax
8.
9.    ; Send EOI
10.   mov al, 20h
11.   out 20h, al
12.
13.   popState
14.
15.   iretq
```

RIP >

P0 ejecuta popState

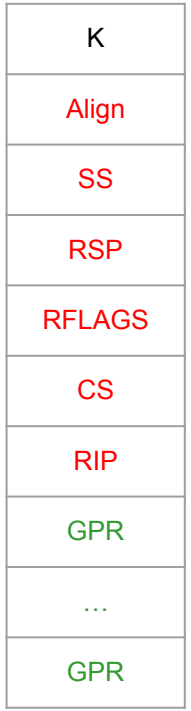


Simulemos con 2 procesos P0 y P1


```
1.  interesting_handler:
2.
3.    pushState
4.
5.    mov rdi, rsp
6.    call schedule
7.    mov rsp, rax
8.
9.    ; Send EOI
10.   mov al, 20h
11.   out 20h, al
12.
13.   popState
14.
15.   iretq
```

RIP >

P0 ejecuta iretq



¿Dónde quedó el RIP? -> ¿Qué ocurre ahora?

Creación de un proceso

Vimos un mecanismo para cambiar entre 2 procesos ya creados, pero…

¿Cómo creamos un proceso y lo incorporamos a este mecanismo? (analogía motor)

¿Cuál es el estado del stack de un proceso que no ejecutó nada aún?

RBP y RSP >

push XXX >

RBP >

RSP >

¿Qué “pinta” debe tener el stack para que funcione con el mecanismo visto?

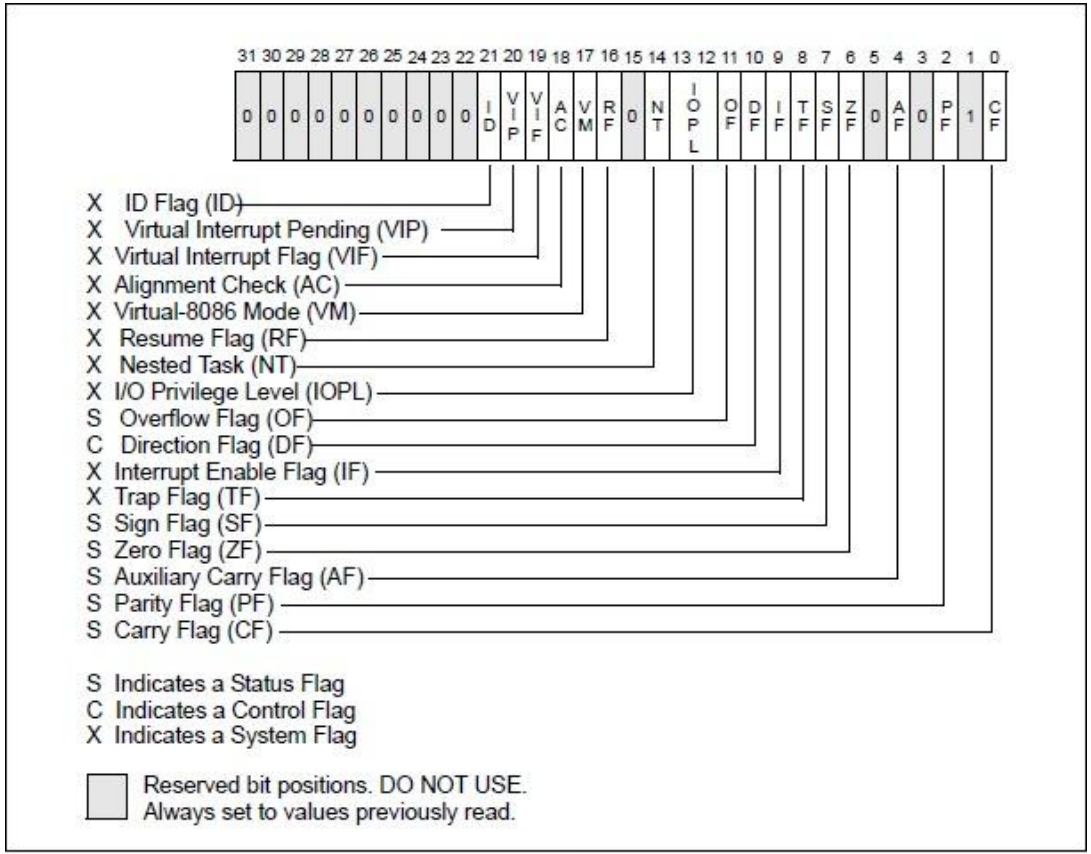
Creación de un proceso - stack

RBP >

Align	¿Qué alineación del RSP tuvo que hacer el hardware cuando “dizque” ocurrió la interrupción? > **???**
SS	¿Cuál era el stack segment cuando “dizque” ocurrió la interrupción? > **0x0**
RSP	¿Cuál era el RSP cuando “dizque” ocurrió la interrupción? > **???**
RFLAGS	¿Cuál era el RFLAGS cuando “dizque” ocurrió la interrupción? > **0x202**
CS	¿Cuál era el code segment cuando “dizque” ocurrió la interrupción? > **0x8**
RIP	¿Cuál era el RIP cuando “dizque” ocurrió la interrupción? > **???**
GPR	¿Cuáles eran los GPR cuando “dizque” ocurrió la interrupción? > **???**
…	¿argc? ¿argv? > ABI calling conventions
GPR	

RSP >

Creación de un proceso - RFLAGS



EFLAGS Register

Creación de un proceso - start

¿Cómo hacemos para que comience a ejecutar este proceso nuevo?

Se lo puede colocar en la lista de procesos con estado READY y forzar una interrupción del timer: **int 0x20**

```
1.  interesting_handler:
2.
3.    pushState
4.
5.    mov rdi, rsp
6.    call schedule
7.    mov rsp, rax
8.
9.    ; Send EOI
10.   mov al, 20h
11.   out 20h, al
12.
13.   popState
14.
15.   iretq
```

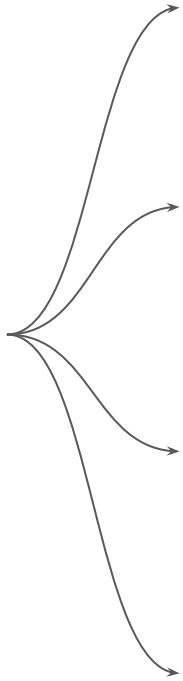
¿Qué stack se está usando al forzar esta interrupción?

¿Qué hacemos con el RSP que llega?

¿Y las syscalls?

¿Por qué “NoRAX”?

```
1.  syscall_handler:
2.
3.    pushStateNoRAX
4.
5.    ; Atender syscall
6.
7.    popStateNoRAX
8.
9.    iretq
```



```
1.  call getpid
2.  ; retornamos normalmente
```

```
1.  call read
2.  ; read determina que se bloquea
3.  ; no podemos retornar (ya)
```

```
1.  call exit
2.  ; no tenemos que volver más
```

```
1.  call yield
2.  ; como read, pero sin bloquear
```

¿Y las syscalls?

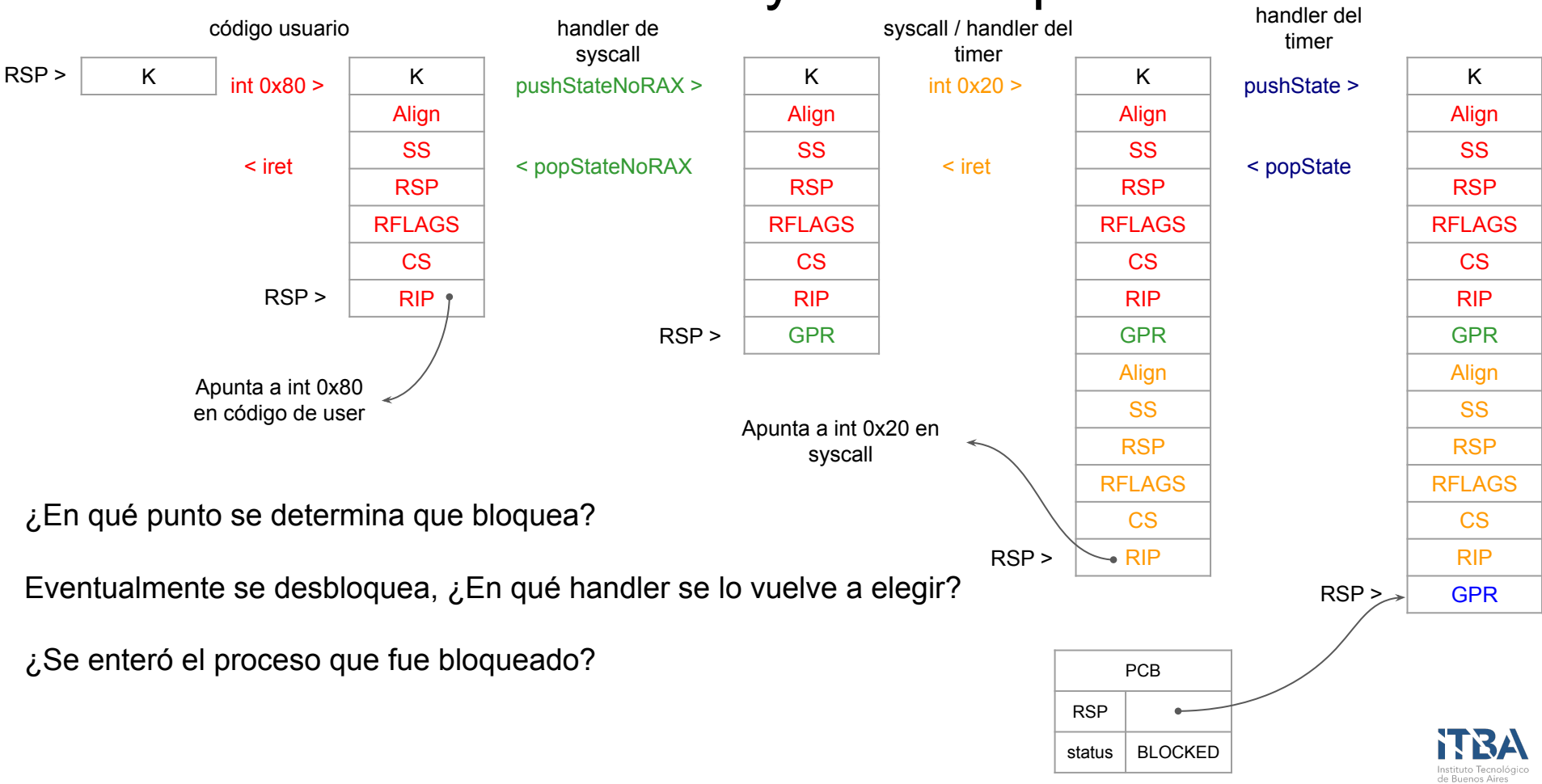
Dentro de la syscall podemos forzar la interrupción del timer como antes

```
1.  call read
2.  ; read determina que se bloquea
3.  ; no podemos retornar (ya)
4.  currentProcess->PCB->state = BLOCKED
5.  int 0x20
```

```
1.  call exit
2.  ; no tenemos que volver más
3.  currentProcess->PCB->state = EXITED
4.  int 0x20
```

```
1.  call yield
2.  ; como read, pero sin bloquear
3.  int 0x20
```

Simulemos una syscall bloqueante



¿En qué punto se determina que bloquea?

Eventualmente se desbloquea, ¿En qué handler se lo vuelve a elegir?

¿Se enteró el proceso que fue bloqueado?

ITBA
Instituto Tecnológico de Buenos Aires

¿Podemos finalizar un proceso con **ret**?

```
1.  int foo(int argc, char *argv[]){
2.
3.    // Does some stuff
4.
5.    return 0;
6.  }
```

¿Cómo está el stack de este proceso justo antes de ejecutar **ret** (return 0)?

¿Ejecutamos **call** foo para ejecutar este proceso?

¿Qué pasa si ejecutamos **ret** de todos modos?

2 opciones:

- Wrapper al estilo _start
- Terminar siempre con exit()

¿Y si no hay procesos READY?

- Esta situación debería ser común ya que al iniciar solo tenemos la shell y esta se bloquea hasta que llegue una interrupción de teclado
- Podemos disponer un proceso cuya única función es ejecutar **hlt**

Resumen

- Switchear entre 2 procesos READY: llamar al scheduler.
- Crear un proceso: simular el stack del caso anterior y llamar al scheduler.
- Finalizar un proceso: cambiar el estado a EXITED y llamar al scheduler, luego se puede borrar el proceso.
- Bloquear un proceso: cambiar el estado a BLOCKED y llamar scheduler, eventualmente algún proceso cambiará su estado a READY.