

Arquitectura de Sistemas

Cambio de hebra

Gustavo Romero López

Updated: 8 de abril de 2025

Departamento de Ingeniería de Computadores, Automática y Robótica

1. Voluntario

1.1 Versión simplificada

1.2 Gestión de la pila

1.3 Viabilidad

2. Involuntario

2.1 Interrupción del reloj

2.2 Gestión de la pila

2.3 Viabilidad

Tanuenbaum Sistemas Operativos Modernos (2.2)

Silberschatz Fundamentos de Sistemas Operativos (4)

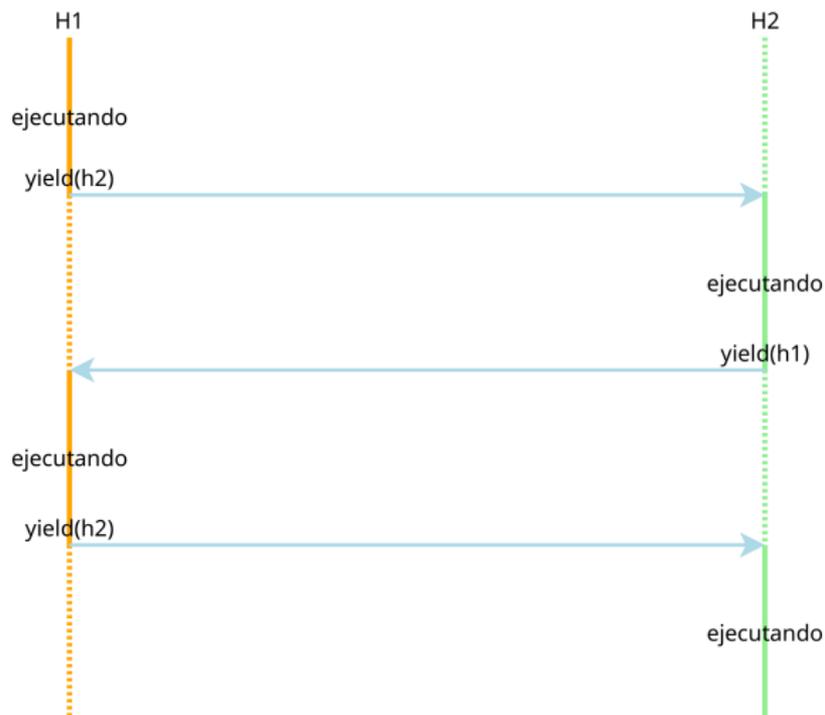
Stallings Sistemas Operativos (4)

- ⊙ Motivos para cambiar de hebra:
 - Comportamiento interno de la hebra.
 - ejemplo: operación de E/S.
 - Objetivos del sistema.
 - ejemplo: planificación justa.
 - las hebras se ejecutan periódicamente $0 < n < \infty$ ciclos.
- ⊙ Tipos de cambio de hebra:
 - Voluntario:
 - planificación cooperativa.
 - entre hebras tipo usuario.
 - gestión mediante funciones.
 - Involuntario:
 - planificación expulsiva (“*preemptive*”).
 - entre cualquier tipo de hebras.
 - gestión mediante llamadas al sistema.

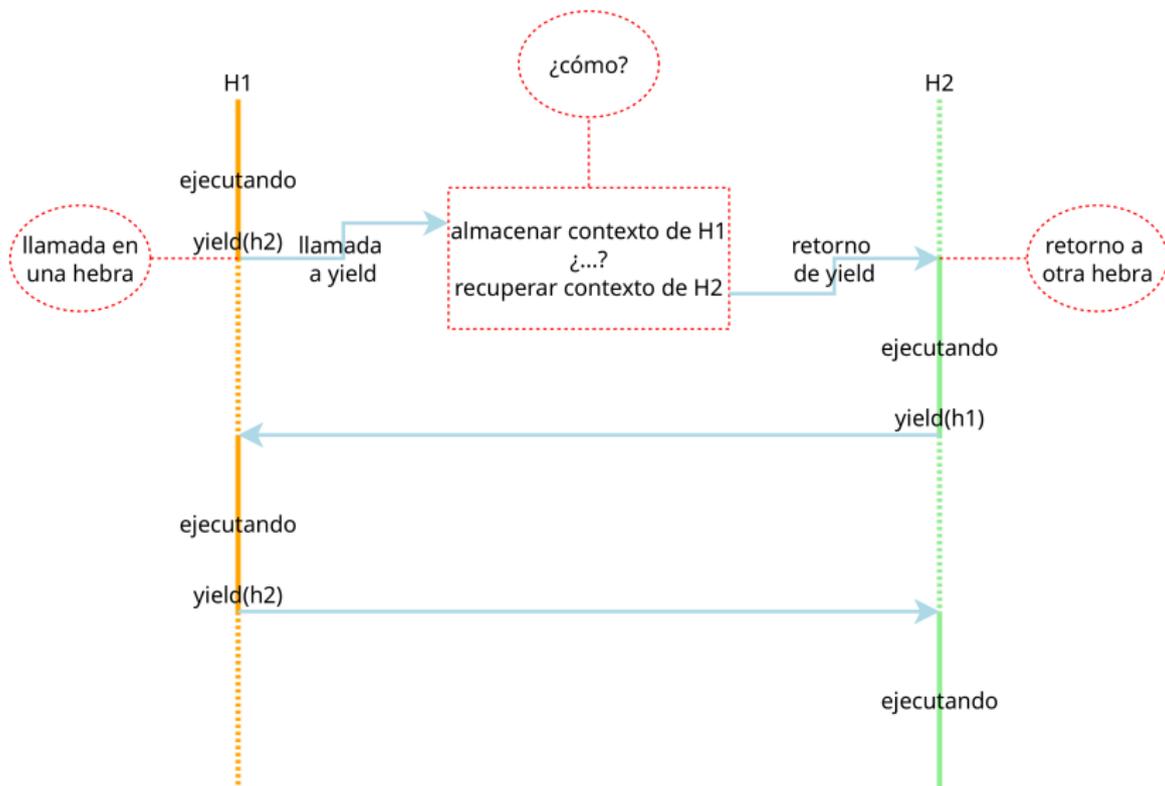
Planificación cooperativa

Modelo simplificado,
supongamos...

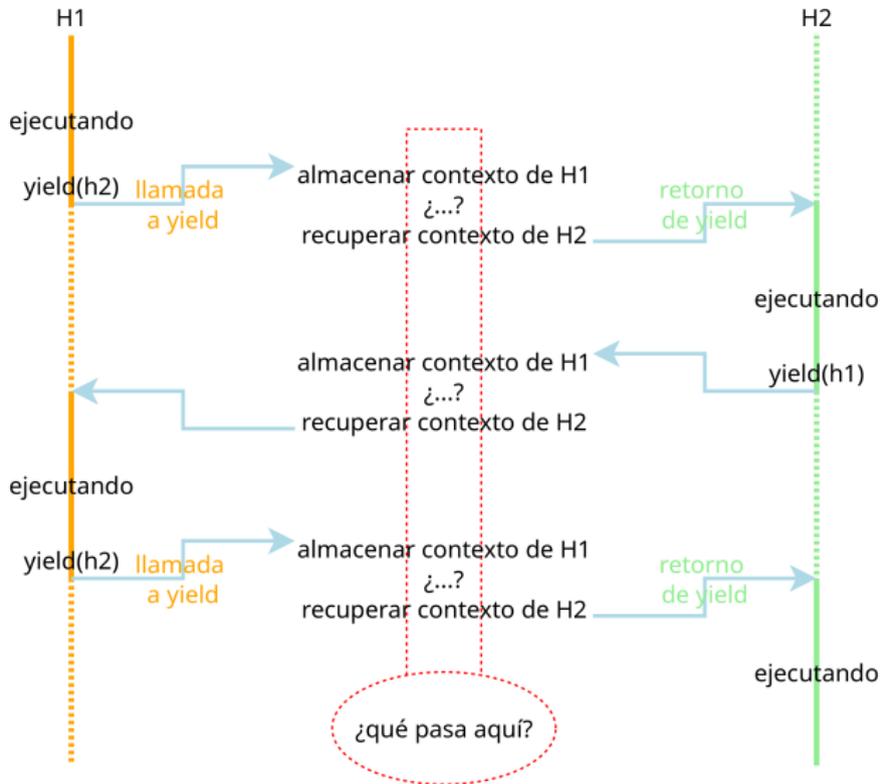
- ⊙ 2 hebras tipo usuario.
- ⊙ limitadas por uso de procesador (sin eventos).
- ⊙ activación alternativa mediante `yield()`.



yield() simplificado (1)



yield() simplificado (2)



yield() simplificado (3)

- ⦿ La primera parte de yield() se ejecuta bajo control de la **hebra llamadora**.
- ⦿ La segunda parte de yield() se ejecuta bajo control de la **siguiente hebra**.
- ⦿ Supondremos que las dos hebras ya han llamado a yield() **antes**.
- ⦿ Cada hebra recibe y cede el control del procesador exactamente en la misma **instrucción**.

yield()

```
void yield(hebra siguiente)
{
    ...almacenar contexto de la hebra actual...
    ¿Qué pasa aquí?
    ...recuperar contexto de la siguiente hebra...
}
```

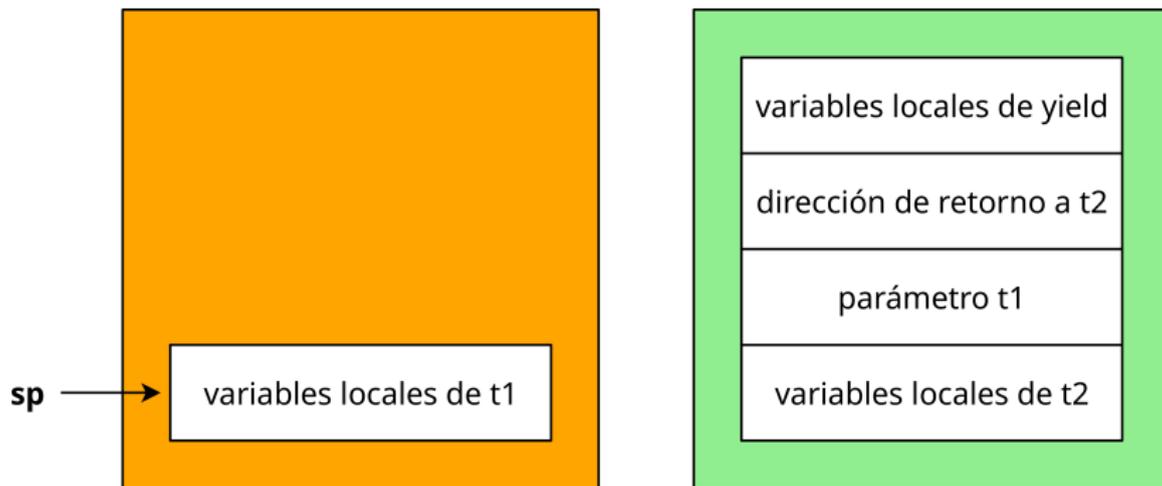
yield() simplificado (4)

- ⊙ Solución: **cambiar el puntero de pila (sp)**.

procedimiento yield()

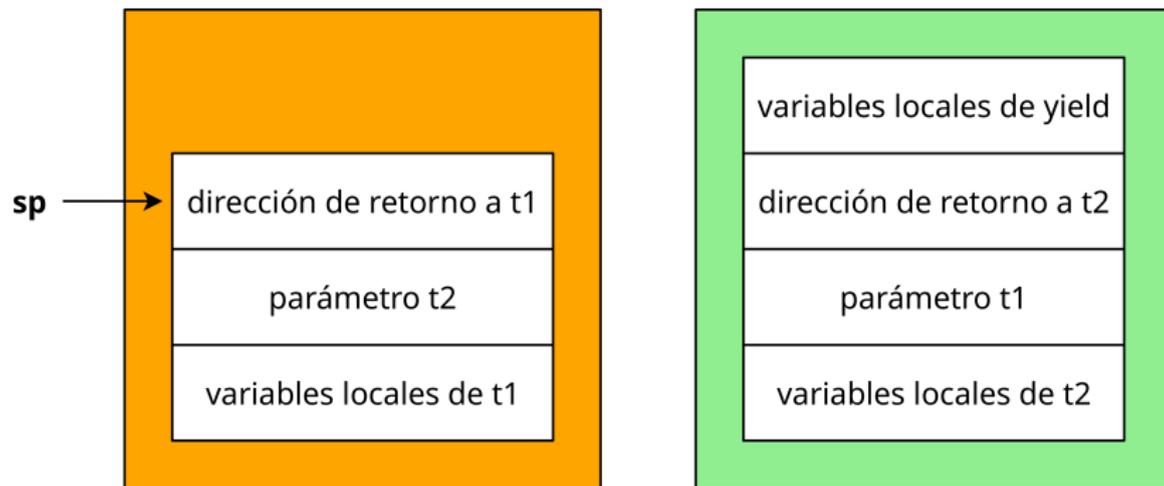
```
void yield(hebra_siguiente)
{
    ...almacenar contexto de la hebra actual...
    hebra_actual.rsp = rsp;    // almacenar rsp
    rsp = hebra_siguiente.rsp; // cambiar rsp
    ...recuperar contexto de la siguiente hebra...
}
```

Contenido de la pila en yield() (1)



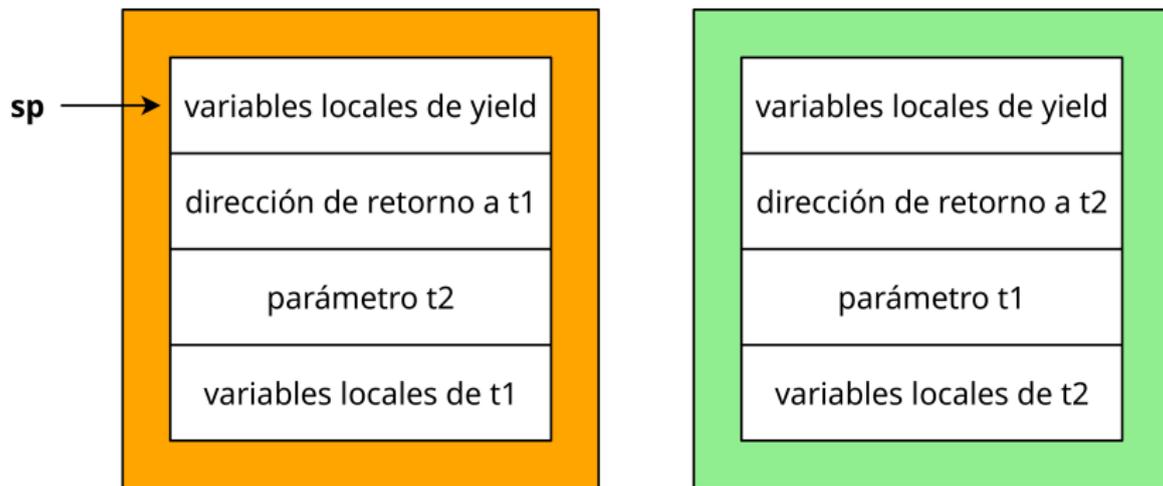
ejecutando t1

Contenido de la pila en yield() (2)



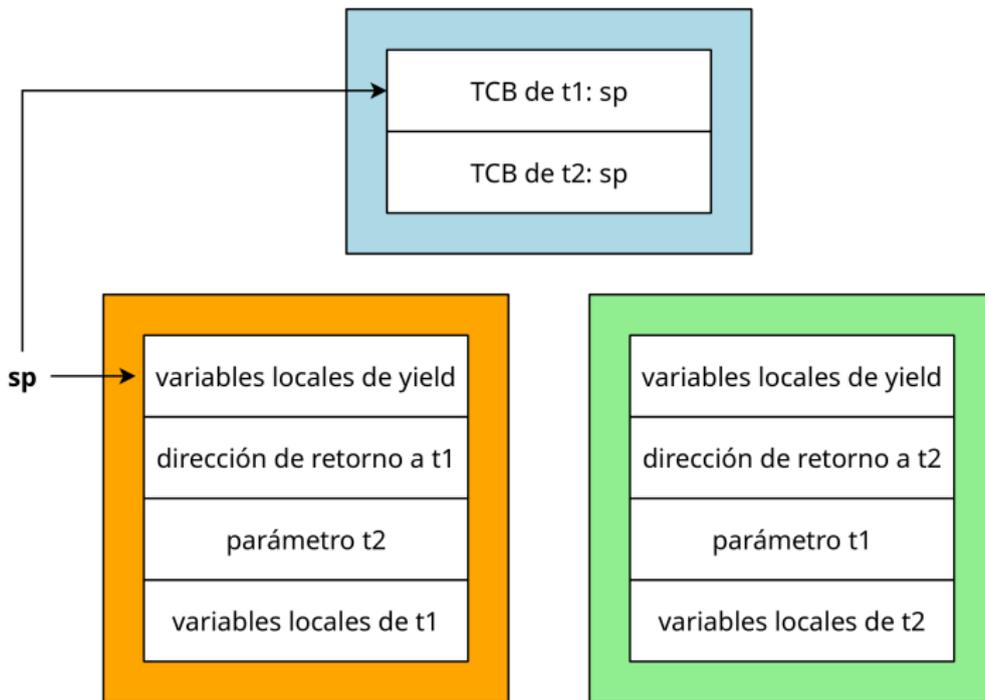
t1 llama a yield

Contenido de la pila en yield() (3)



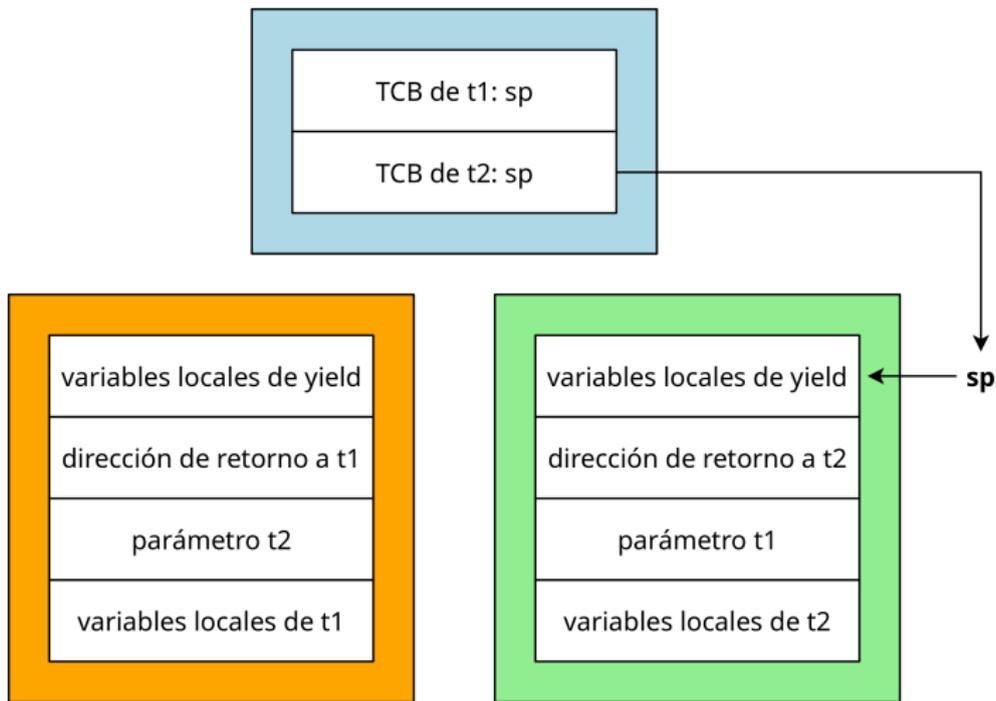
t1 almacena su estado

Contenido de la pila en yield() (4)



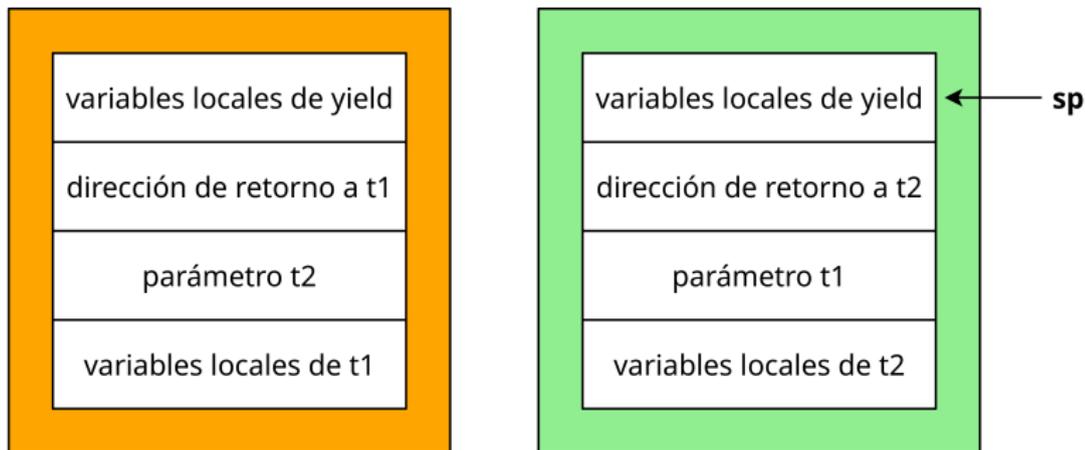
t1 almacena el puntero de pila en su TCB

Contenido de la pila en yield() (5)



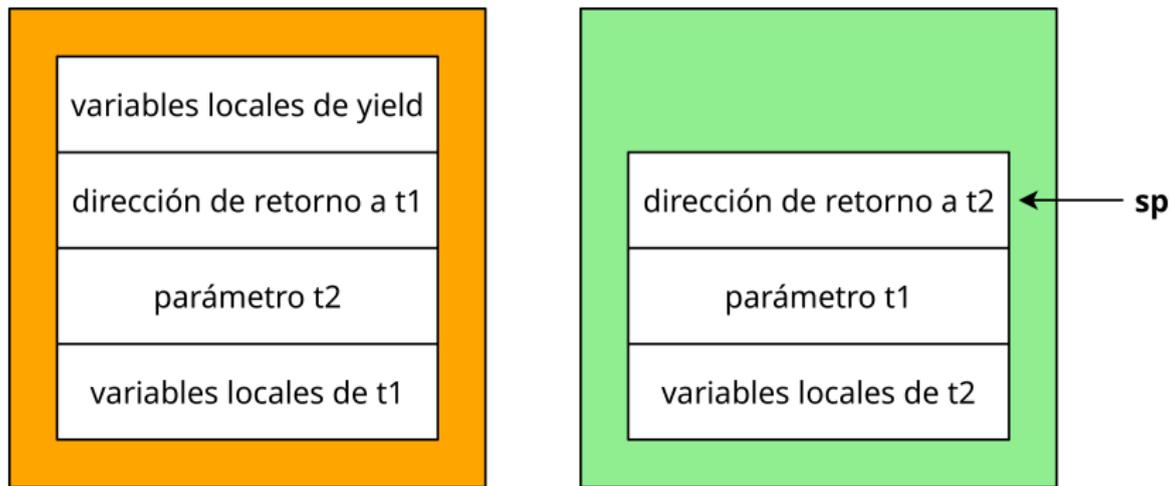
t1 recupera el puntero de pila de t2 de su TCB

Contenido de la pila en yield() (6)



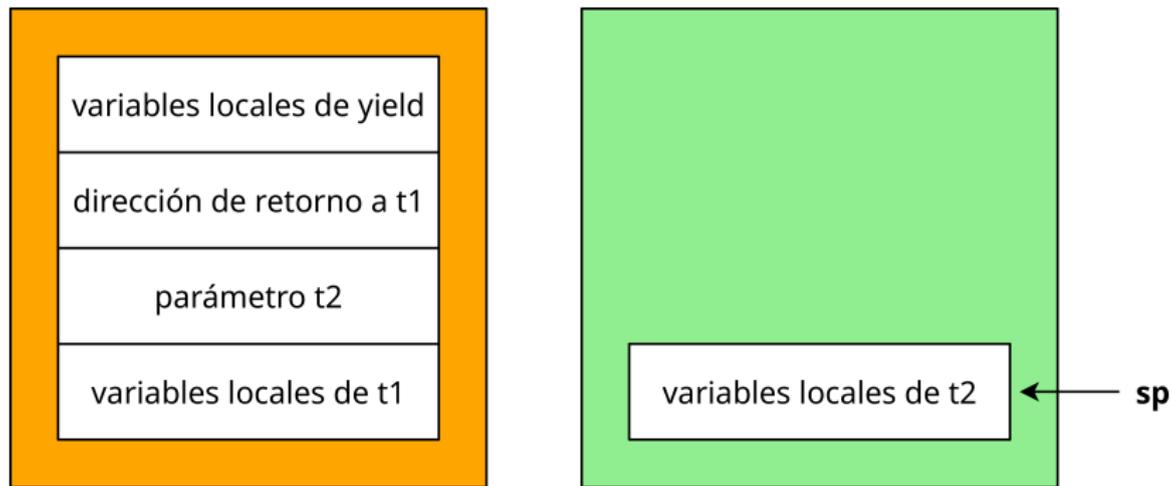
t2 reinicia su ejecución desde donde la abandonó en una llamada anterior a yield

Contenido de la pila en yield() (7)



t2 retorna de yield

Contenido de la pila en yield() (8)

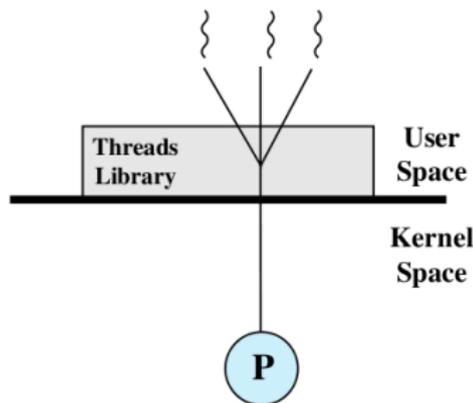


ejecutando t2

Resumen sobre hebras de usuario y yield()

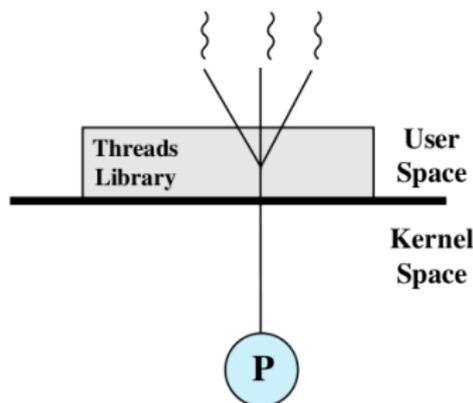
Suponiendo que hay un solo procesador y `yield()` es la única posibilidad de activación...

- ⦿ solo la pila de la hebra en ejecución es utilizable.
- ⦿ el número y orden de las pilas es conocido.
- ⦿ el contenido de las pilas difiere muy poco.
- ⦿ cada hebra puede tener diferentes variables locales.



Restricciones de yield() para hebras de usuario

- ⊙ El **núcleo** no es **consciente** de la existencia de hebras tipo usuario.
- ⊙ La gestión de hebras se realiza en la aplicación o en un subsistema mediante una **biblioteca de hebras**.
- ⊙ El cambio de hebra no requiere privilegios de modo núcleo \implies ahorro de cambios de proceso y modo.
- ⊙ La **planificación** puede ser **específica** por cada proceso.
- ⊙ ¿Qué sucede cuando una hebra **no** llama a yield()? \implies ☠



- ⊙ Creación y destrucción de hebras.
- ⊙ Paso de mensajes entre hebras.
- ⊙ Planificación de hebras.
- ⊙ Sincronización de hebras.
- ⊙ Almacenamiento y restauración del estado de una hebra.

Posible trabajo para el núcleo

- ⊙ Aunque el núcleo no es consciente de la existencia de hebras tipo usuario gestiona el funcionamiento del **proceso** que las contiene.
- ⊙ Ejemplo:
 - Cuando una hebra realiza una **llamada al sistema** \implies el núcleo **bloquea** al **proceso** completo.
 - Desde el punto de vista del planificador de hebras de usuario la hebra sigue en estado **ejecutando**.
- ⊙ Los estados de las hebras son **independientes** del estado del proceso que las contiene.

Ventajas e inconvenientes de las hebras de usuario

Ventajas:	Inconvenientes:
la gestión de las hebras no requiere del núcleo \Rightarrow no requiere cambio de proceso \Rightarrow mayor velocidad	si una hebra se bloquea \Rightarrow se bloquean todas las hebras del proceso
la política de planificación puede ser específica por aplicación \Rightarrow utilizar la más conveniente	el núcleo solo puede asignar el procesador a procesos \Rightarrow 2 hebras del mismo proceso nunca podrán ejecutarse simultáneamente \Rightarrow solo conurrencia
las hebras de usuario pueden utilizarse en cualquier SO de forma fácil mediante el uso de una biblioteca de hebras	

Inconvenientes de un `yield()` de sistema

Si es tan rápido, ¿por qué no utilizarlo en el SO?

1. La planificación cooperativa requiere de usuarios cooperativos.
 - ¿Dejarías libre un recurso que podrías necesitar más adelante?
2. Incluso entre usuarios cooperativos, el problema de colocar `yield()` de manera óptima es complejo.
3. Muchos eventos requieren cambios de modo que el núcleo debe gestionar.
4. Existe desde siempre... `sched_yield()`.

síncronas

- ⊙ la hebra actual termina.
- ⊙ llamada al sistema (E/S).
- ⊙ paso de mensaje: enviar mensaje/esperar respuesta.
- ⊙ hebra cooperativa llama a `yield()`.

asíncronas

- ⊙ la hebra excede su fracción de tiempo asignado.
- ⊙ la hebra en ejecución posee una prioridad inferior a otra preparada.
- ⊙ un controlador de dispositivo solicita ser atendido (evento).

Excepciones (eventos síncronos):

- ⊙ Fallos (eventos reproducibles):
 - división entre cero (durante la instrucción).
 - violación de espacio de direcciones (durante la instrucción).
- ⊙ Eventos impredecibles:¹
 - fallo de página (antes de la instrucción).
- ⊙ Puntos de ruptura ("*breakpoint*"):
 - datos (después de la instrucción).
 - código (antes de la instrucción).
- ⊙ Llamada al sistema.

¹¿Cuándo será predecible un fallo de página? → política de paginación local.

Interrupciones (eventos asíncronos):

⊙ Reloj:

- final de la fracción de tiempo asignado.
- señal de despertar (alarma).

⊙ Impresora:

- atasco de papel.
- tinta agotada.

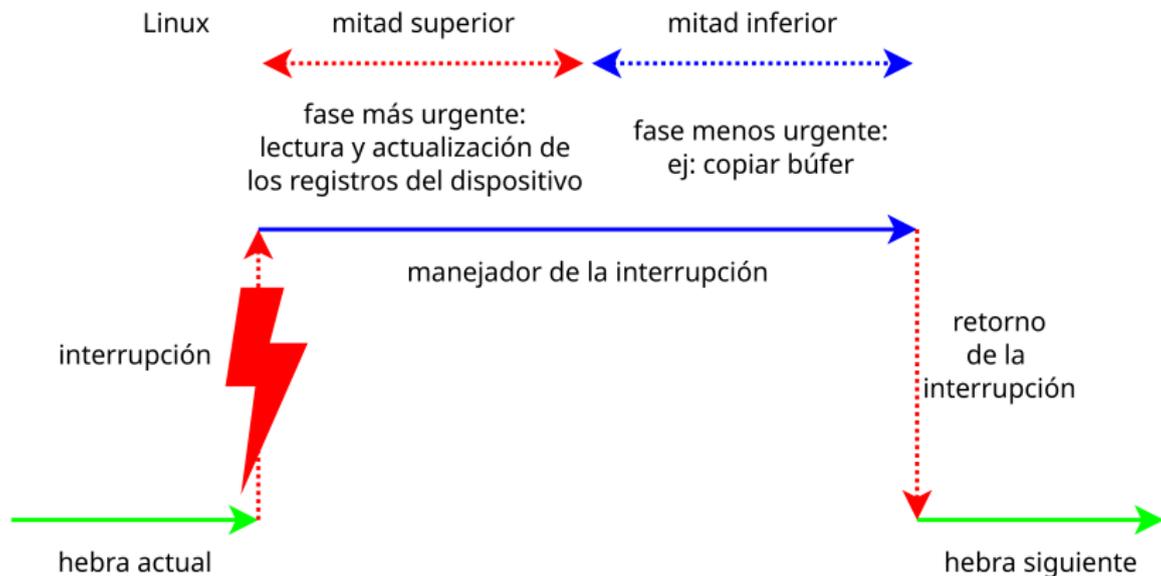
⊙ Red:

- paquete recibido.

⊙ Otros procesadores:

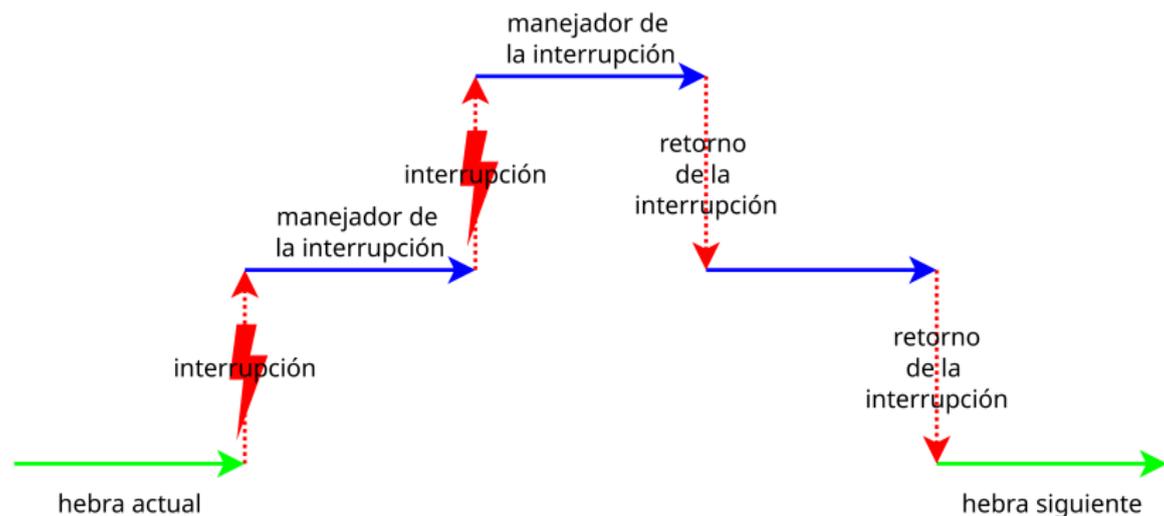
- señales entre procesadores.

Gestión de interrupciones en Linux



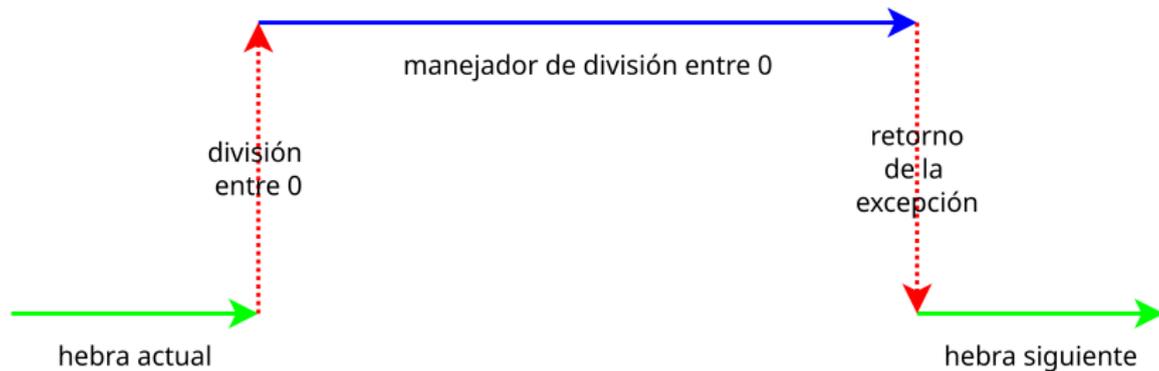
- ⊙ mitad superior no interrumpible, mitad inferior si.
- ⊙ la hebra siguiente puede ser la misma que la actual.

Interrupciones anidadas



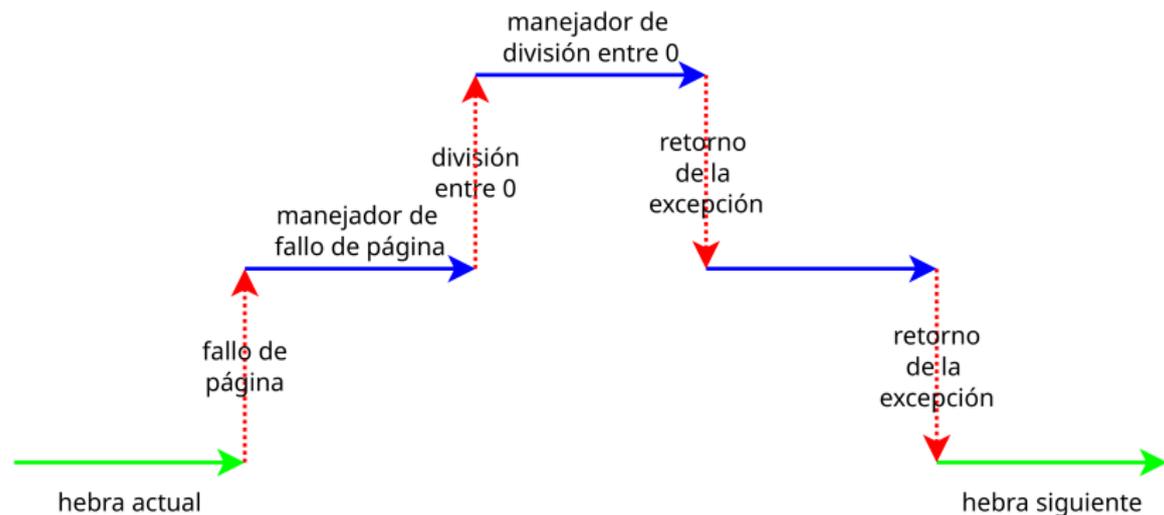
- ⊙ Las interrupciones pueden anidarse sin problema.
- ⊙ Un circuito se encarga de la gestión de interrupciones
 - Intel 8259 PIC → APIC.

Gestión de excepciones



- ⊙ Pueden escogerse manejadores de excepciones específicos.

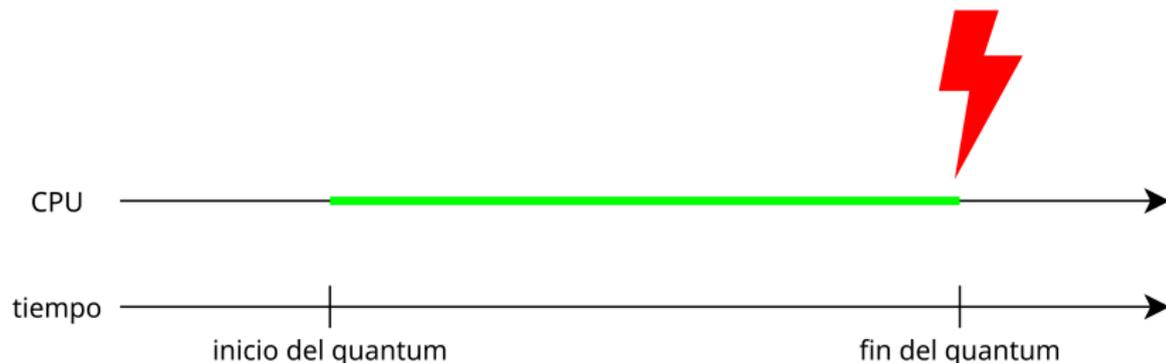
Excepciones anidadas



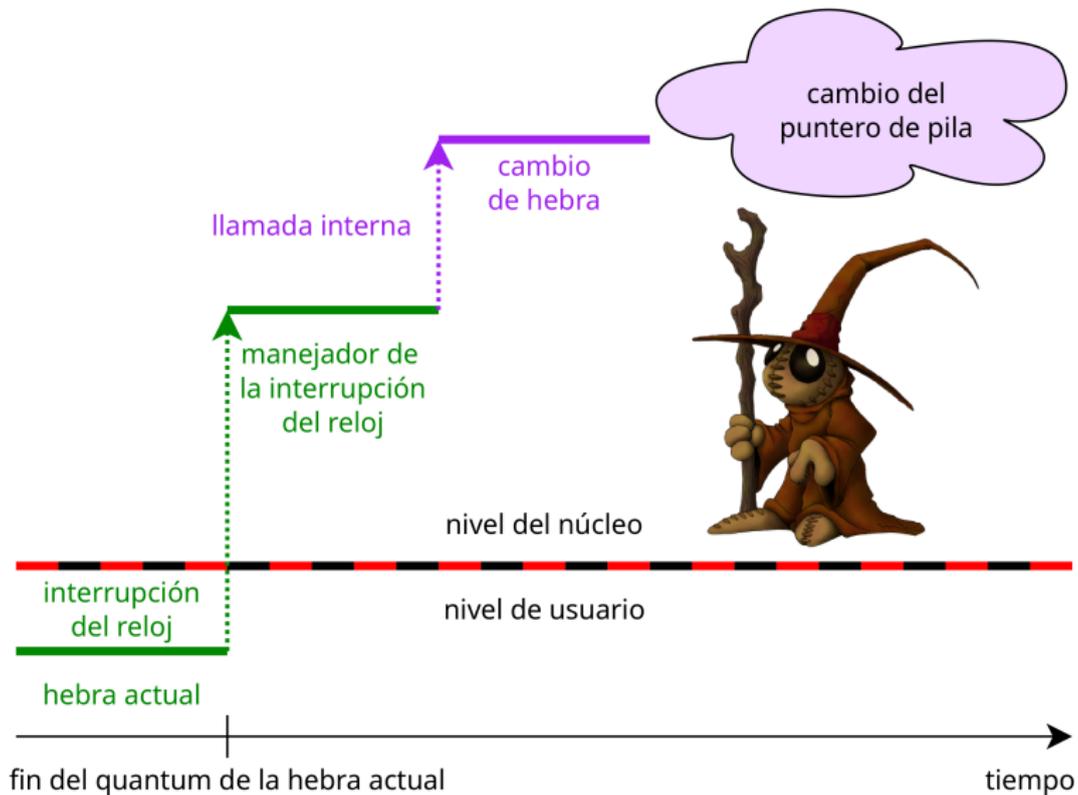
- ⊙ Algunos sistemas permiten anidar 2 ó 3 excepciones pero no más.
- ⊙ Suelen indicar un fallo en el propio sistema operativo.

Finalización del tiempo asignado

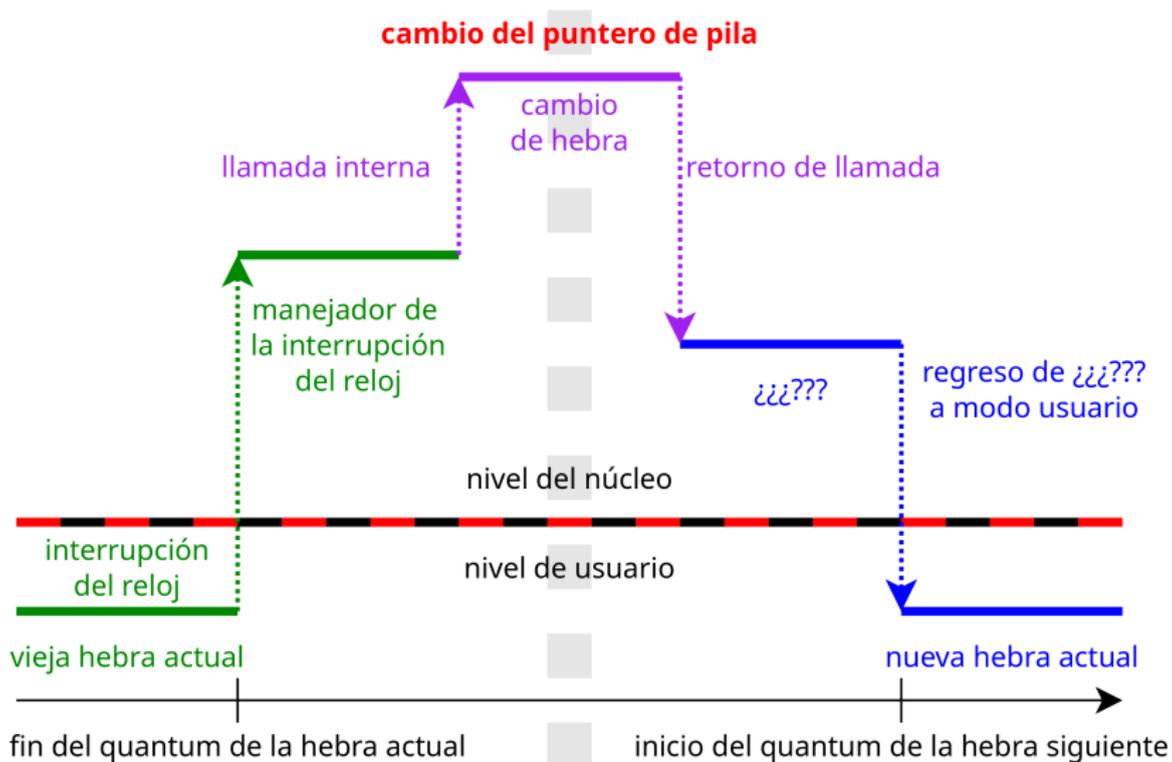
- ⊙ El caso más habitual de cambio de hebra se produce al agotar su fracción de tiempo asignada “**quantum**”.
- ⊙ Hebras tipo **núcleo**.
- ⊙ Objetivo más frecuente: planificación justa.
- ⊙ Versión simplificada de la interrupción de reloj.



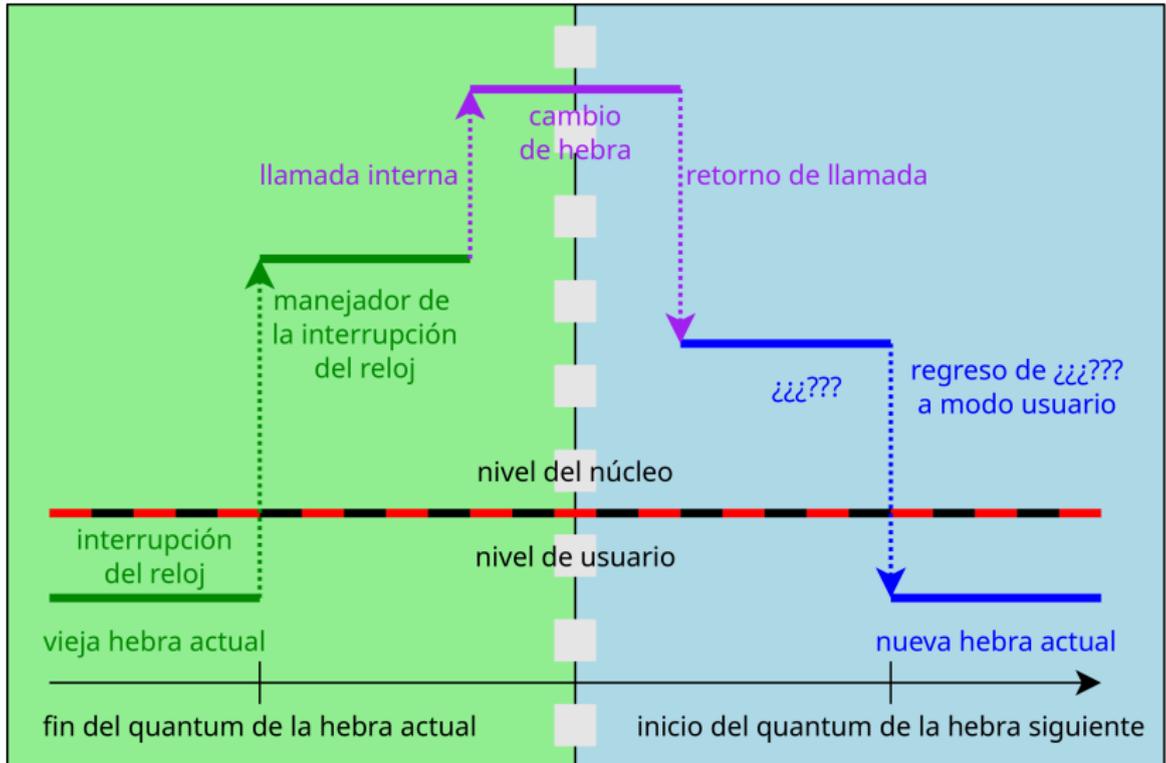
cambio de hebra simplificado (1)



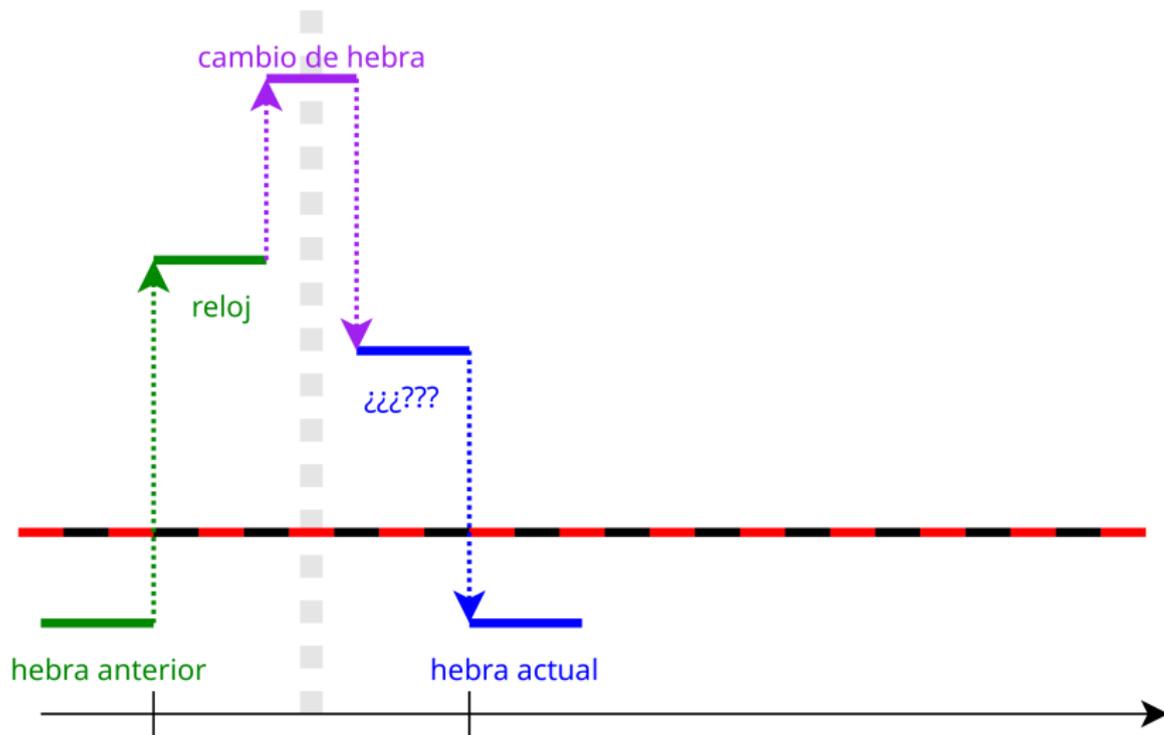
cambio de hebra simplificado (2)



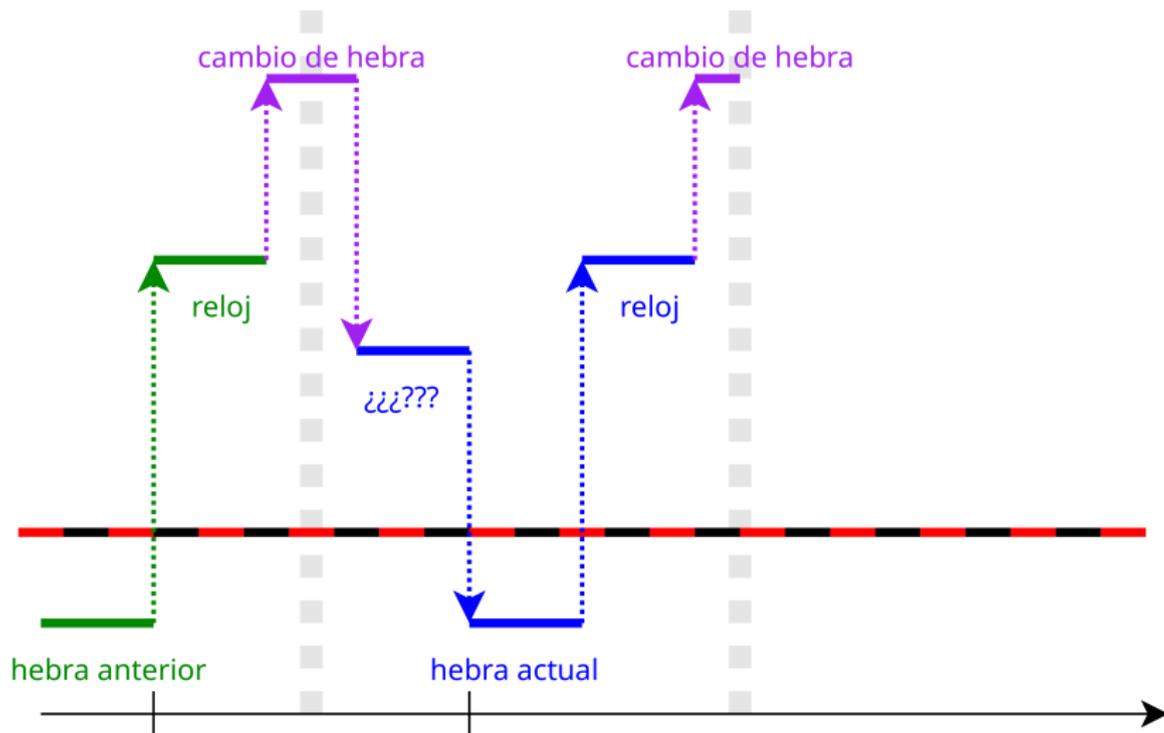
cambio de hebra simplificado (3)



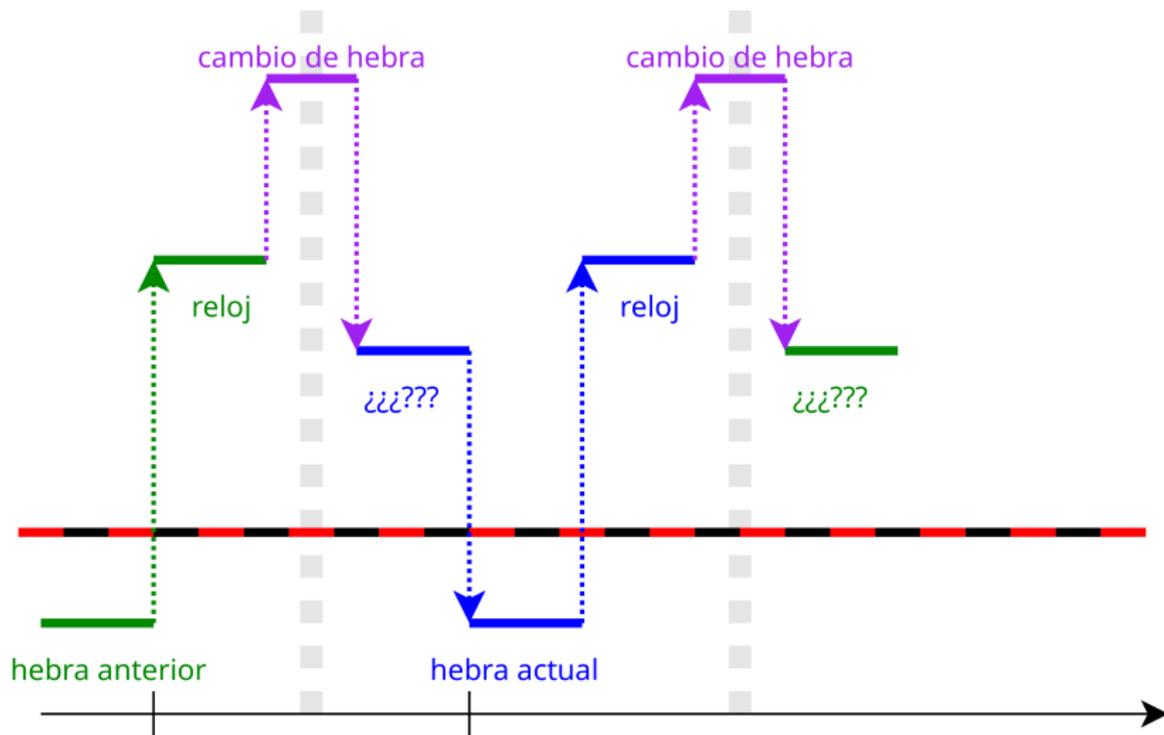
cambio de hebra simplificado (4)



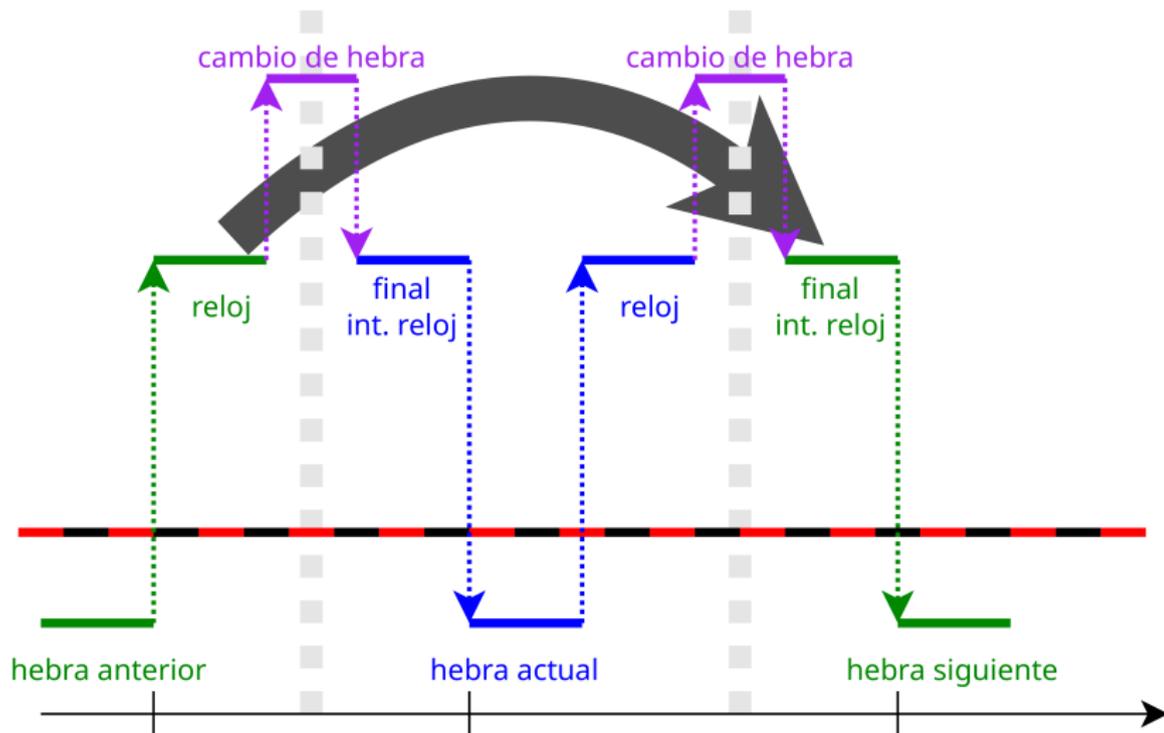
cambio de hebra simplificado (5)



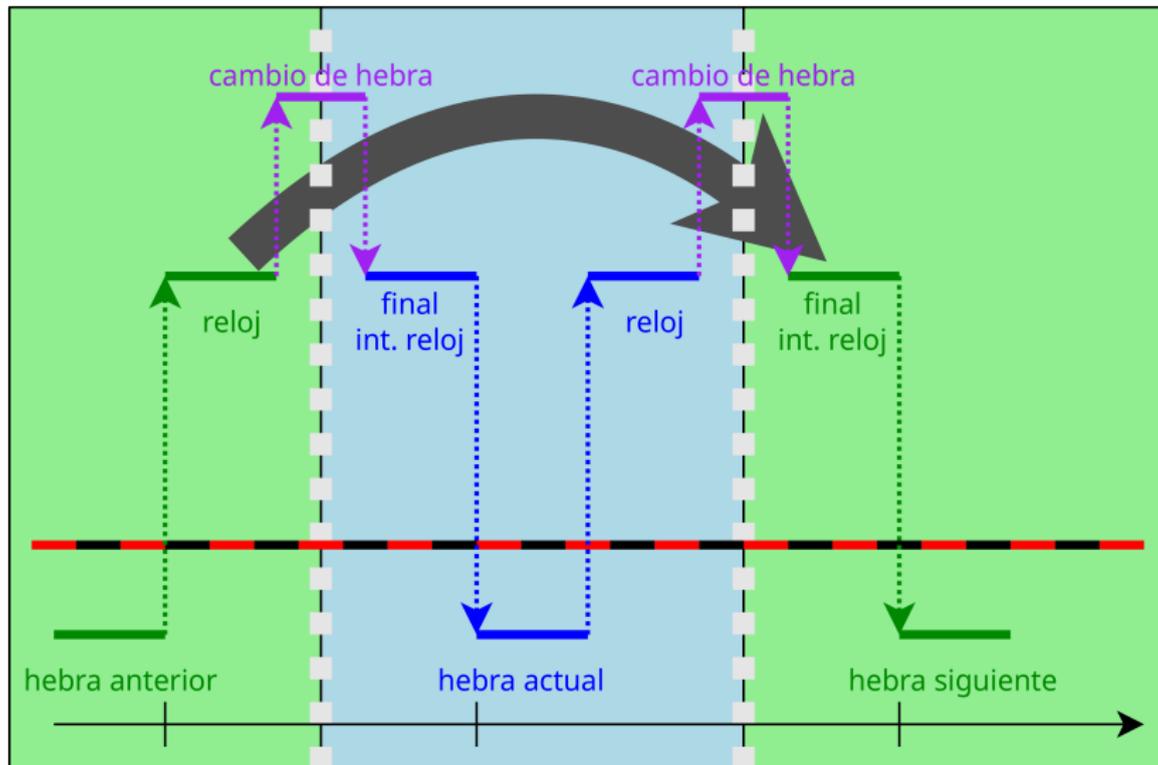
cambio de hebra simplificado (6)



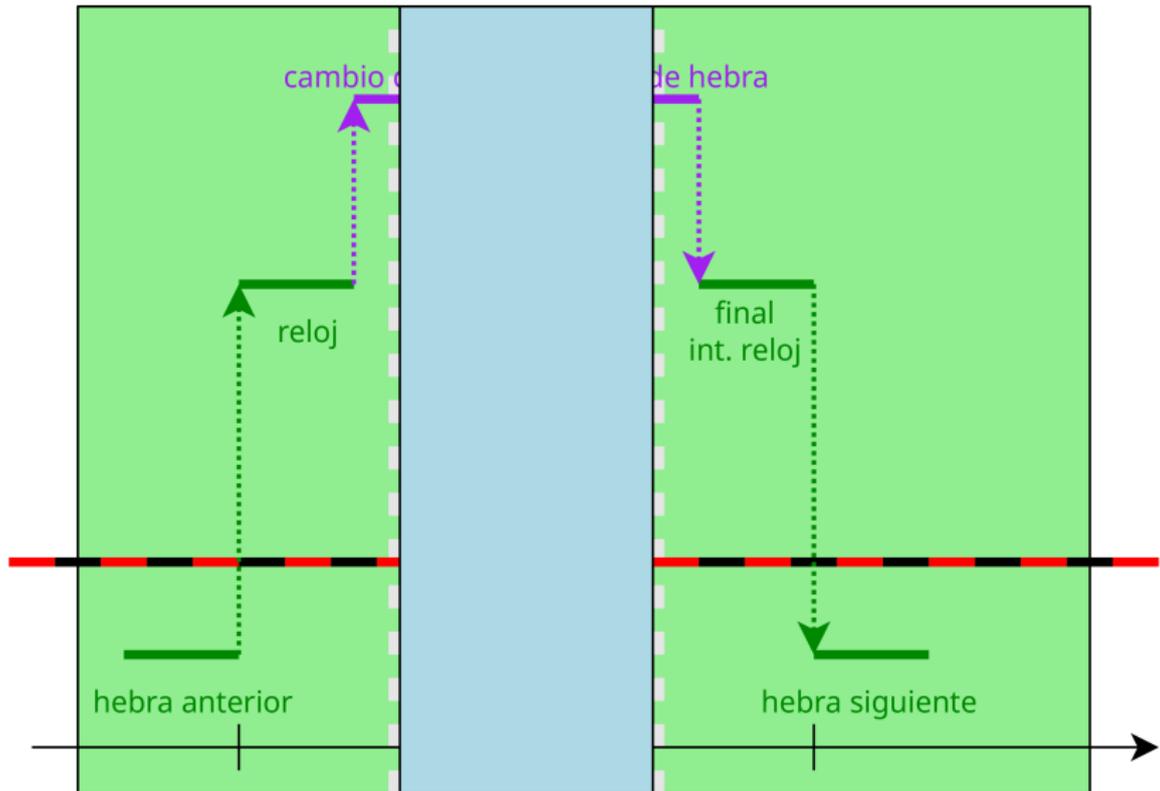
cambio de hebra simplificado (7)



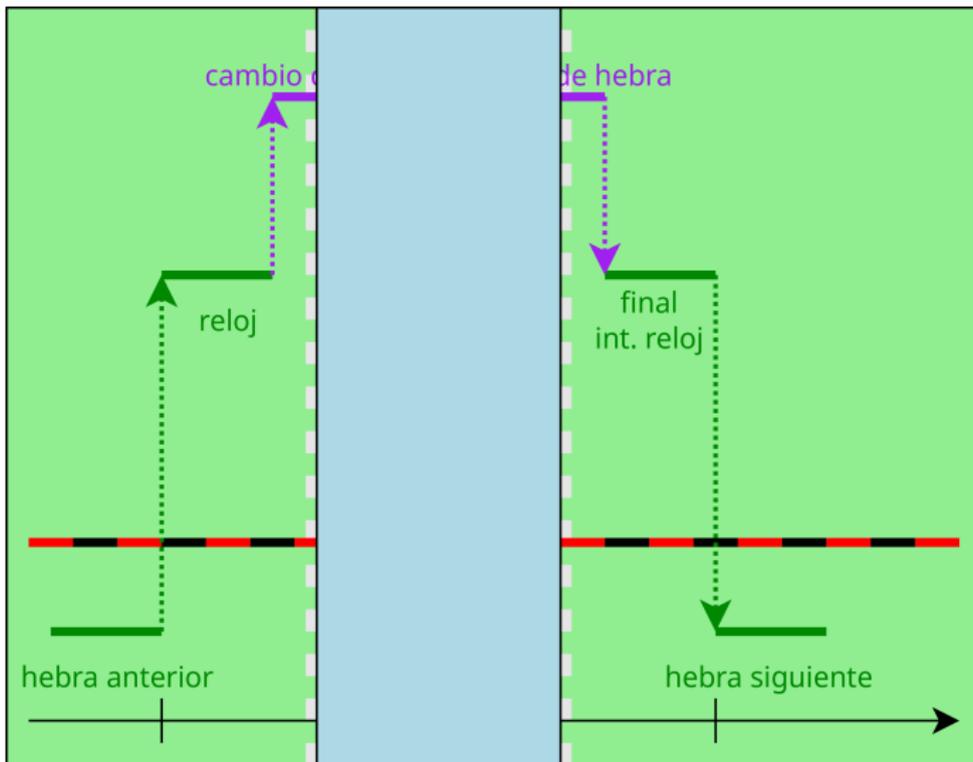
cambio de hebra simplificado (8)



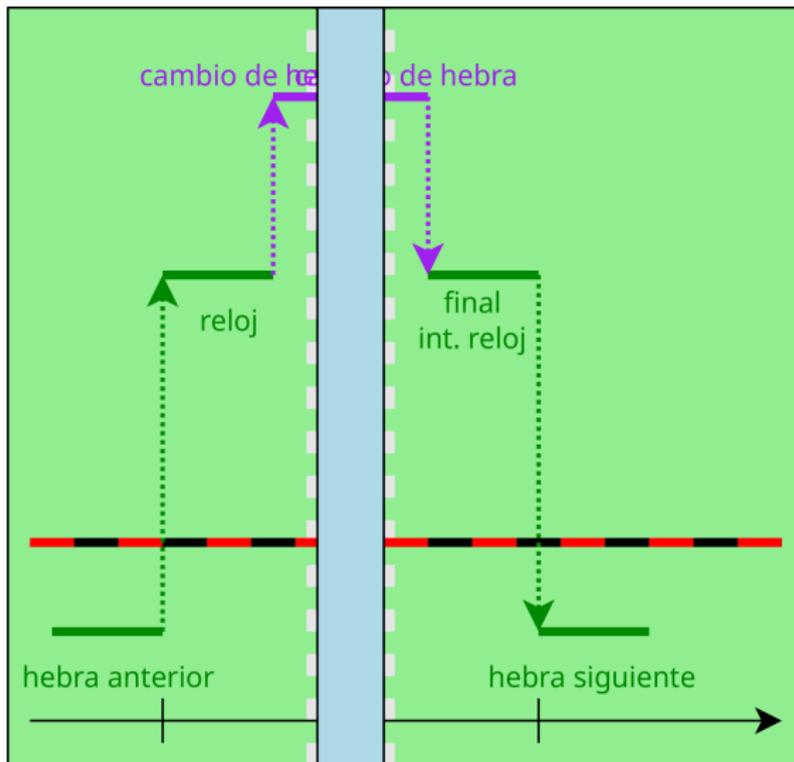
cambio de hebra simplificado (9)



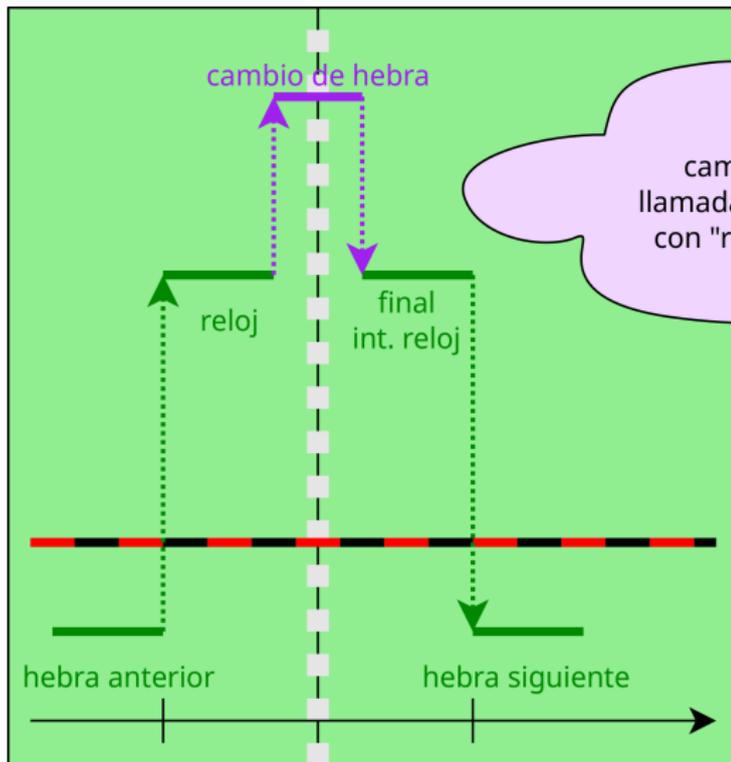
cambio de hebra simplificado (10)



cambio de hebra simplificado (11)



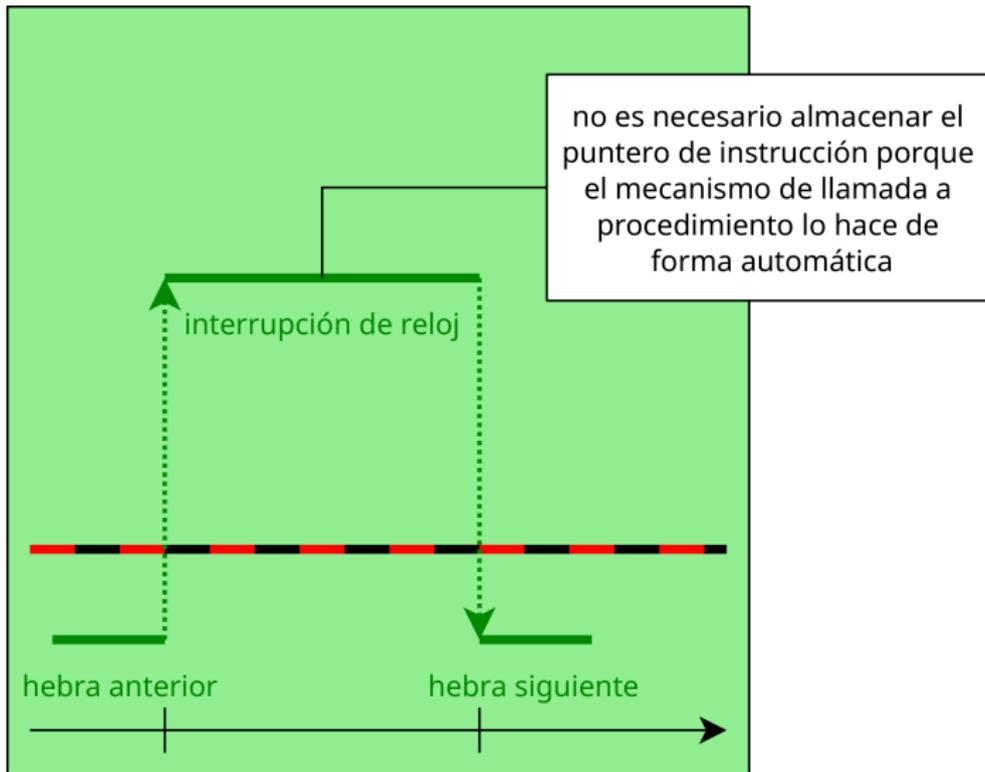
cambio de hebra simplificado (12)



cambio de hebra =
llamada a procedimiento
con "retorno perezoso"

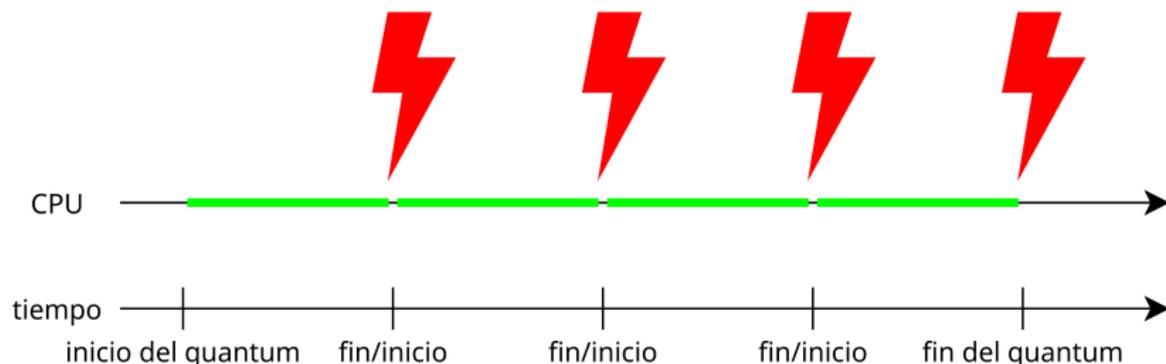


cambio de hebra simplificado (13)

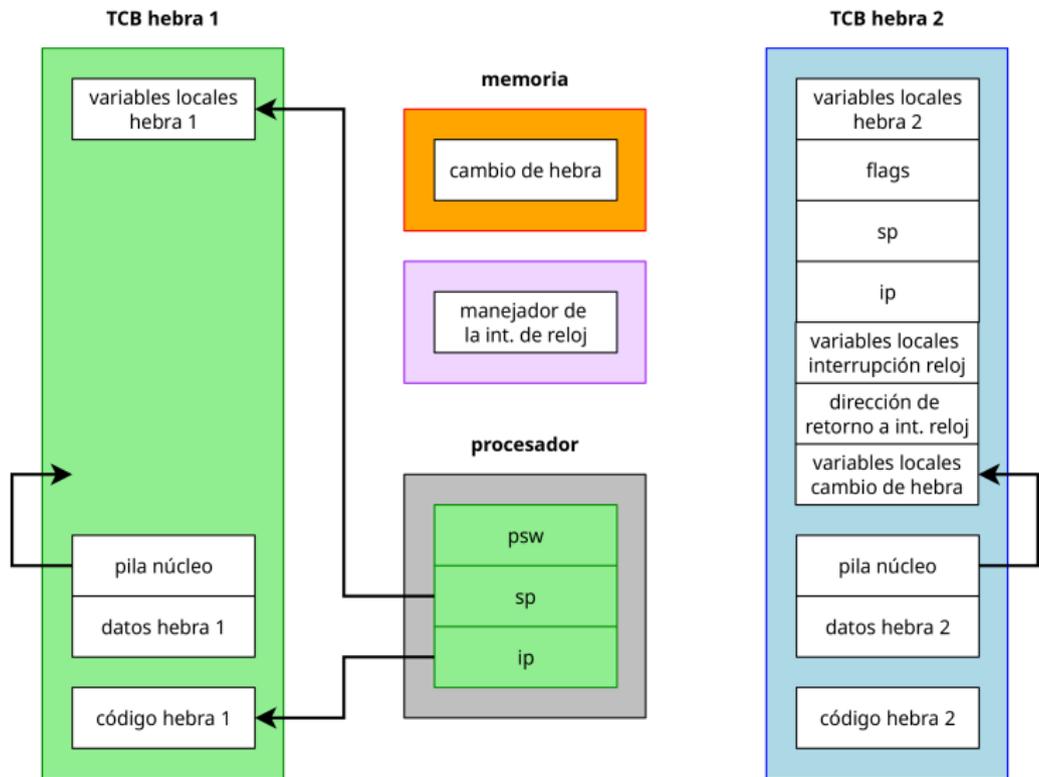


cambio de hebra simplificado (14)

- ⊙ El hardware automáticamente almacena `ip`, `sp` y las banderas en la **pila núcleo** de la hebra actual (x86).
- ⊙ La pila núcleo forma parte del TCB de cada hebra.

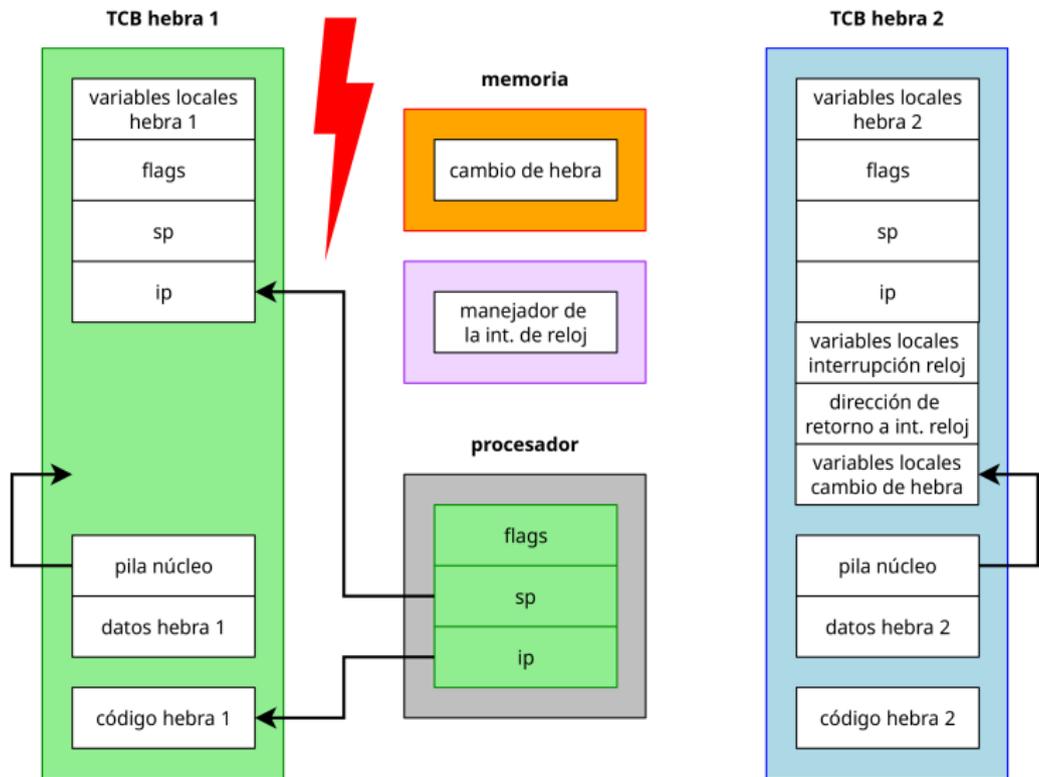


Estudio de la pila (1)



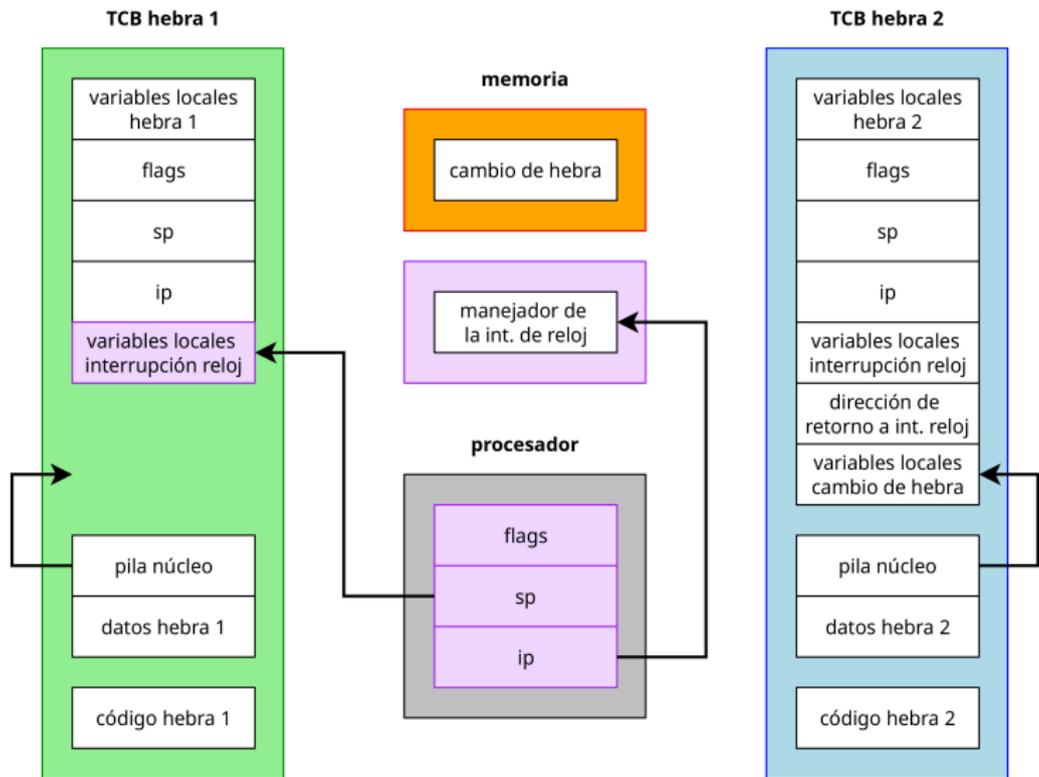
la hebra 1 se está ejecutando

Estudio de la pila (2)



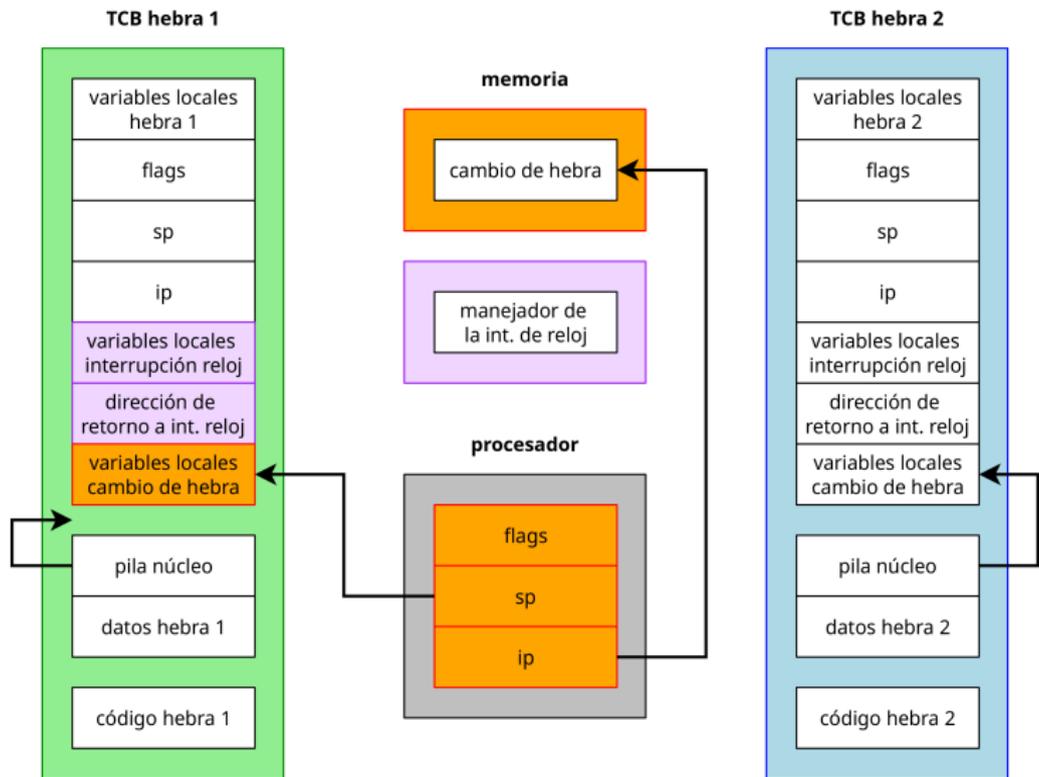
la interrupción de reloj almacena el estado de la hebra 1 y ...

Estudio de la pila (3)



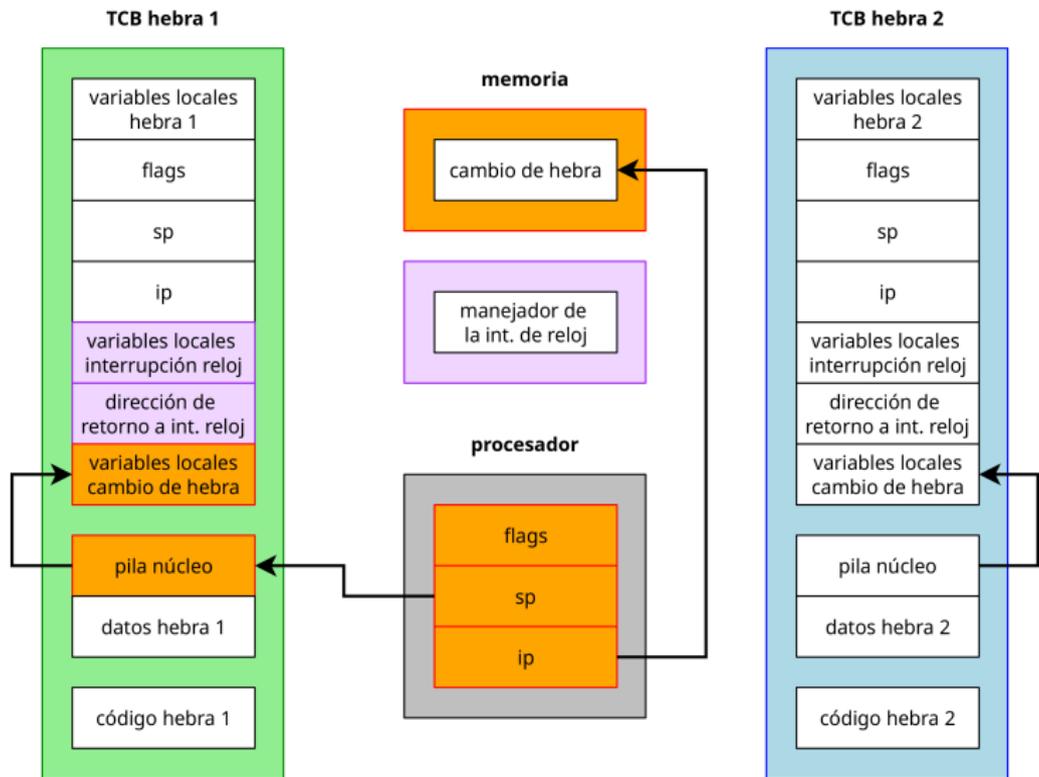
... carga el contexto del manejador de la interrupción de reloj

Estudio de la pila (4)



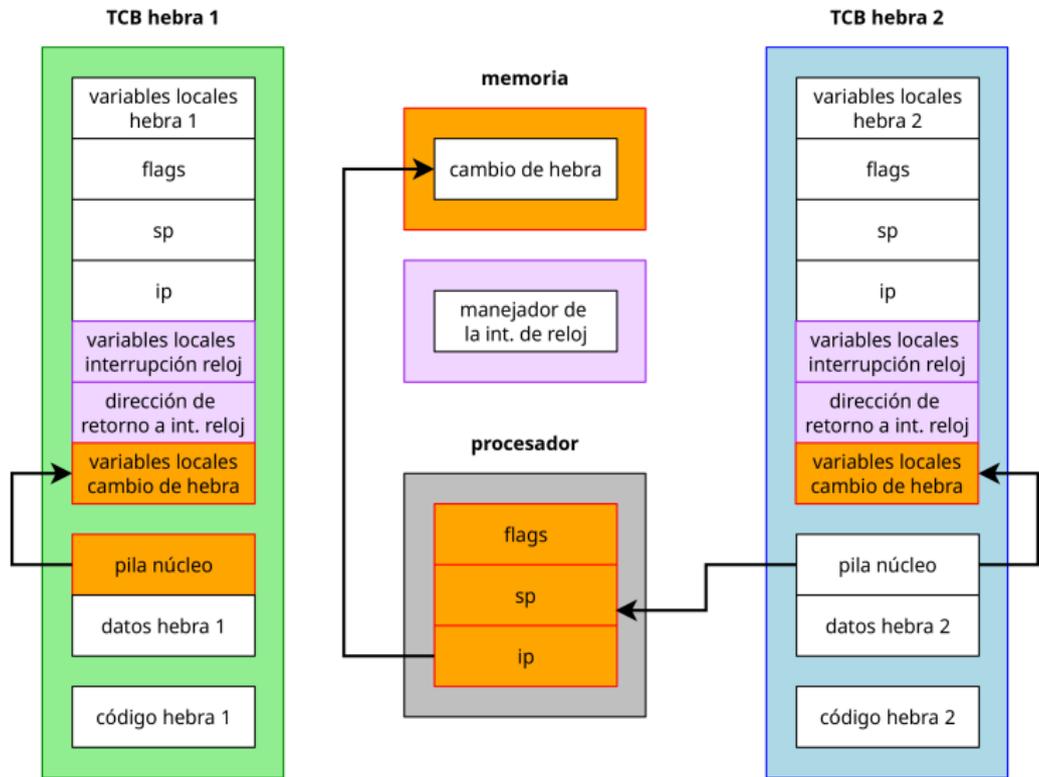
el manejador de la interrupción de reloj decide el fin de la hebra 1 y llama a cambio de hebra

Estudio de la pila (5)



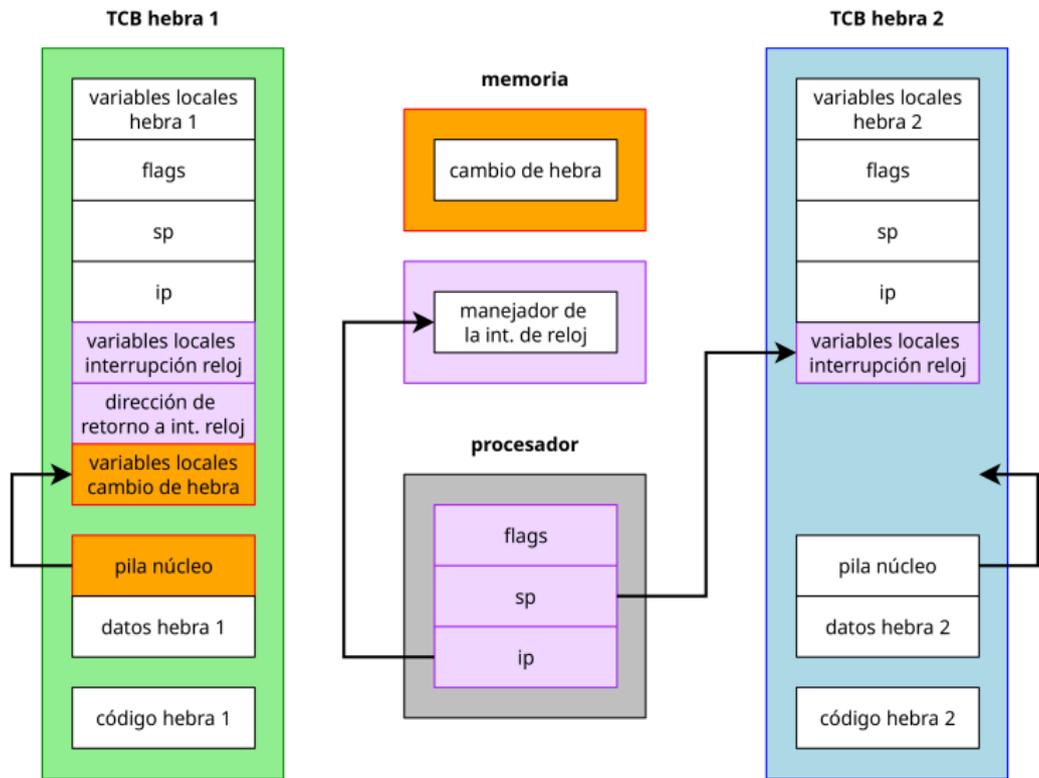
el núcleo almacena el puntero de pila de la hebra 1 en su TCB y ...

Estudio de la pila (6)



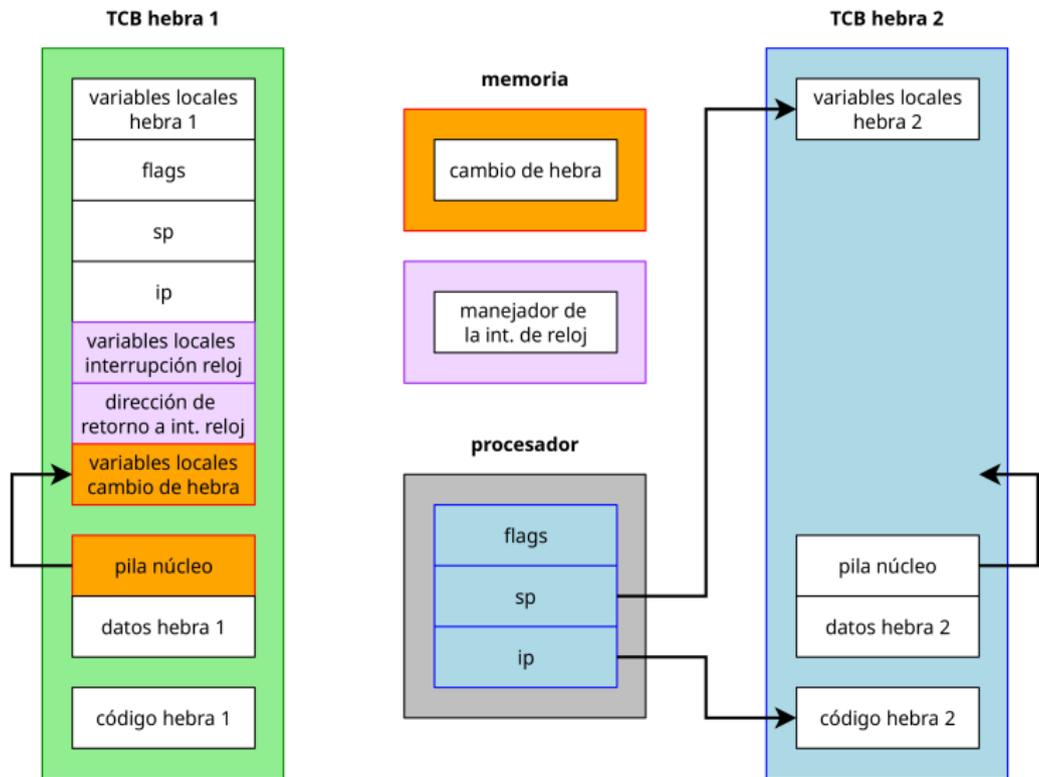
... y recupera el puntero de pila de la hebra 2 de su TCB

Estudio de la pila (7)



finalizar la ejecución de cambio de hebra y retornar al manejador de la interrupción de reloj

Estudio de la pila (8)



finalizar la ejecución del manejador de la interrupción de reloj y retornar a la hebra 2

... en función del número de pilas núcleo utilizadas:

- ⦿ Una única pila núcleo para todas las hebras.
- ⦿ Una pila núcleo por hebra.

¿Qué implicaciones tienen estas dos soluciones?

El estado del núcleo es almacenado de forma implícita en la pila de activación del núcleo.

- ⊙ Si la hebra debe bloquearse mientras está en modo núcleo, simplemente podemos cambiar a la pila de otra hebra hasta su reanudación.
- ⊙ La reanudación es tan sencilla como intercambiar de nuevo la pila por la original.
- ⊙ La **expulsión** ("preemption") es fácil de implementar.
- ⊙ No existe diferencia conceptual entre modo núcleo y modo usuario

- ⊙ ¿Cómo puede una única pila núcleo dar soporte a múltiples hebras? \iff ¿Cómo manejar las llamadas al sistema bloqueantes
 - **Continuaciones** (“*continuations*”): Draves et al. Using continuations to implement thread management and communication in operating systems. Proc.13th SOSP (1991).
 - **Núcleo sin estado** (“*stateless kernel*”): Ford et al. Interface and execution models in the fluke kernel. Proc.3th OSDI.
- ⊙ ¿Puede una única pila núcleo dar soporte a múltiples hebras en un sistema multiprocesador?
 - No \implies es necesaria una **pila por procesador**.

Problemas que hemos ignorado por simplicidad (1)

- ⊙ Si la hebra siguiente no es conocida por adelantado... ¿cómo decidimos que hebra ejecutar a continuación? \implies planificador.
- ⊙ Una vez escogida la hebra siguiente, ¿de dónde recuperamos su TCB?
- ⊙ Si la pila núcleo es parte del TCB \implies corremos el peligro de que el desbordamiento de la pila lo destruya.
- ⊙ ¿Cómo lograr la inicialización y finalización de una hebra?

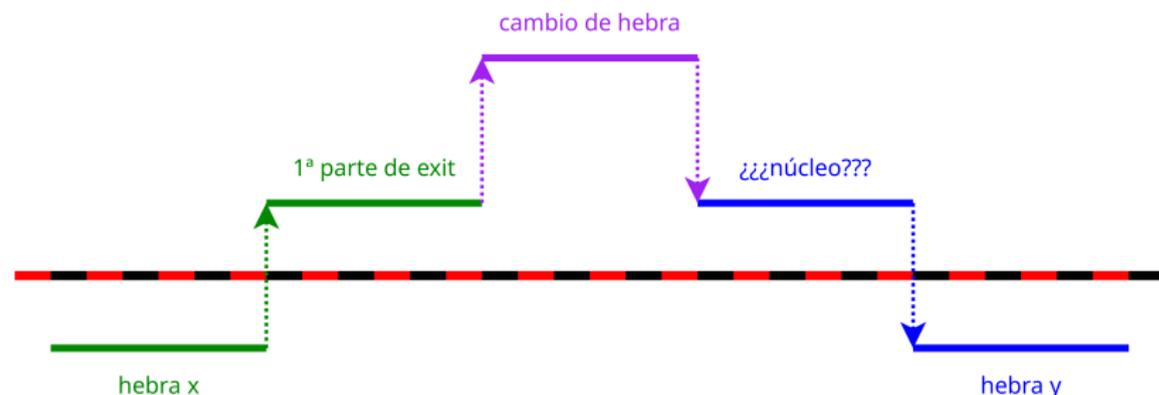
Problemas que hemos ignorado por simplicidad (2)

- ⊙ El tamaño de la pila núcleo no es un problema en la práctica.
- ⊙ El desbordamiento de la pila núcleo supone la existencia de error dentro del núcleo.
- ⊙ En cuanto a la inicialización y finalización de una hebra... como principio general es deseable... “RESOLVER LOS CASOS ESPECIALES EMPLEANDO LAS SOLUCIONES DE LOS CASOS GENERALES”.

Finalización de una hebra

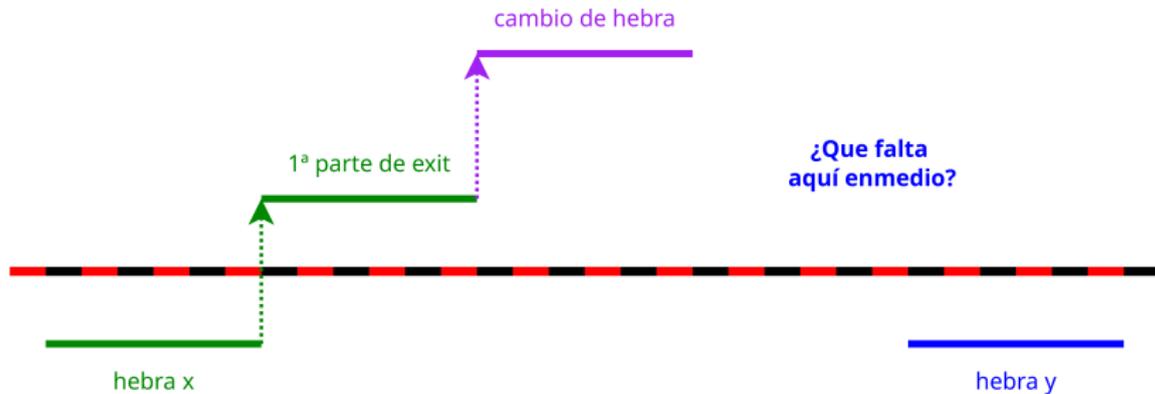
¿Cómo finalizar una hebra?

- ⊙ Hacer todo el trabajo necesario para limpiar el entorno de la hebra.
- ⊙ Cambiar a otra hebra y **nunca regresar** a la hebra que finaliza.
- ⊙ Sería deseable no requerir mecanismos adicionales.



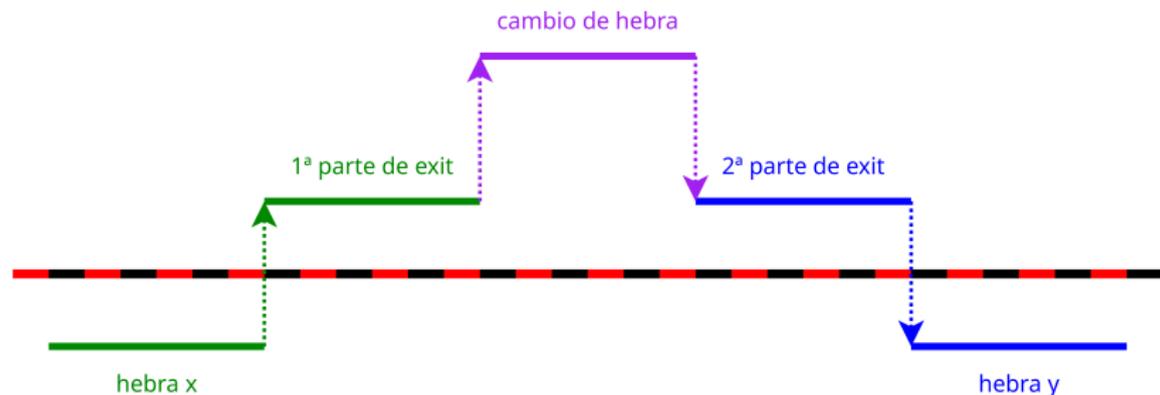
Inicialización de una hebra

¿Qué hacer cuando cambiamos a una nueva hebra por **primera vez**?



Inicialización de una hebra

- ⊙ **Inicializar la pila núcleo** de la nueva hebra con la segunda parte de las funciones `cambio_de_hebra()` y `exit()`.
- ⊙ El retorno desde `cambio_de_hebra()` cede el control a la segunda parte de `exit()`.
- ⊙ El regreso desde `exit()` al modo usuario provoca la ejecución de la primera instrucción de la nueva hebra.

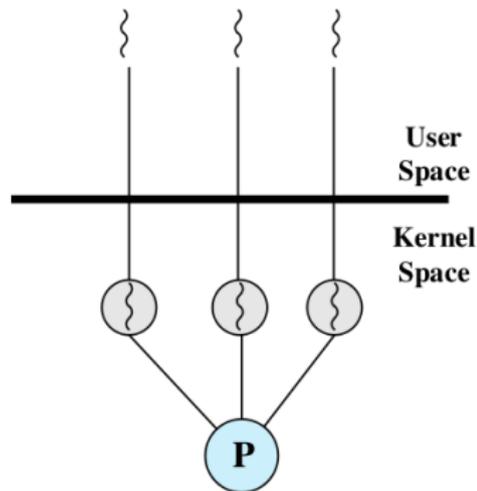


El problema del procesador desocupado

- ⊙ ¿Qué hacer cuando no hay otra hebra a la que cambiar?
- ⊙ Solución: evitar el problema introduciendo la siempre ejecutable **hebra ociosa** → vista en la práctica 2.
- ⊙ ¿Cómo garantizar que una hebra sea siempre ejecutable?
 - evitar cualquier posible evento.
- ⊙ ¿Qué otras propiedades tiene la hebra ociosa?
 - ahorro de energía → hlt.
 - lanzar tareas periódicas de baja prioridad → balanceador de carga, desfragmentador,...
- ⊙ ¿Cuándo introducirla en el sistema?
 - antes de arrancar.
 - durante el arranque.
 - después de arrancar.

Resumen hebras tipo núcleo

- ⊙ La gestión se solicita mediante llamadas al sistema y ejecuta código en el núcleo.
- ⊙ No es necesaria una biblioteca de hebras sino utilizar la API de núcleo.
- ⊙ El núcleo mantiene los TCBs de las hebras.
- ⊙ El cambio de hebra requiere la intervención del núcleo \implies cambio de contexto (modo + proceso).
- ⊙ La planificación se realizará sobre hebras y no sobre procesos.



Ventajas e inconvenientes de las hebras tipo núcleo

Ventajas:	Inconvenientes:
Varias hebras de un proceso pueden ejecutarse simultáneamente sobre varios procesadores ⇒ paralelismo real	El cambio de hebra puede costar hasta 2 cambios de proceso y modo .
Una llamada al sistema solo bloquea a la hebra llamadora y no al resto de hebras del proceso	La sobrecarga provocada por la intervención del núcleo hará que su rendimiento sea peor que las hebras tipo usuario
El núcleo puede implementarse de forma multihebra	

Notas sobre implementación

Hebras tipo usuario:

- ⊙ muchas soluciones: coroutines(C++20), boost::fiber
- ⊙ getcontext() y setcontext() + sigaltstack()
- ⊙ sigsetjmp() y siglongjmp() + sigaltstack()
- ⊙ setjmp() y longjmp() + sigaltstack()
- ⊙ todo desde 0 → mala idea pero se aprende mucho :)

Hebras tipo núcleo:

- ⊙ requiere soporte por parte del sistema operativo
- ⊙ la más extendida es pthread
- ⊙ disponible en casi todo lenguaje de programación, ej:
std::**thread**

Ejemplos: <https://pccito.ugr.es/as/practicas/longjmp>