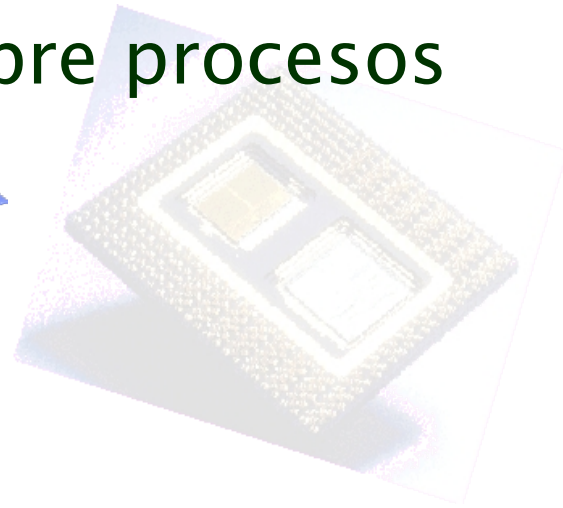
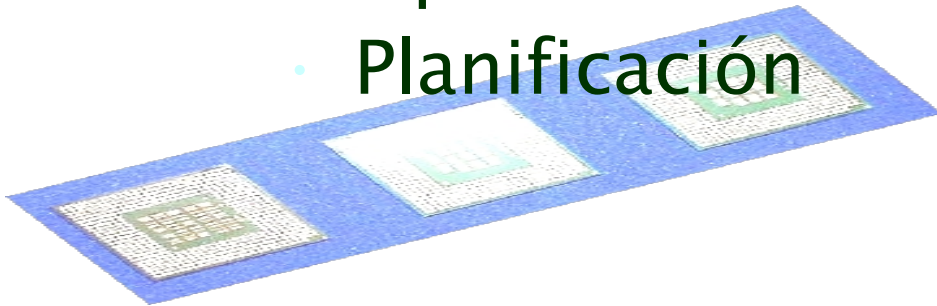


2

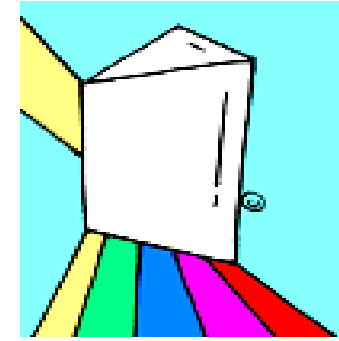
hebras

- La abstracción proceso
- Hebras y procesos ligeros
- Operaciones sobre procesos
- Planificación





El argumento de la simplicidad



- El mejor modo de resolver un problema complejo es dividirlo en subproblemas más pequeños.
- En un SO, y otros programas complejos, existen actividades que ocurren a la vez: 10 solicitudes de disco, 2 mensajes de red, 7 pulsaciones de teclas, 20 aplicaciones, ...
- Para programar tal sistema:
 - (a) un programa que lo gestiona todo.
 - (b) aislar cada actividad en un proceso.



El bucle del SO

- Podríamos construir el SO como:

```
for (;;) {  
    if (aplicación())      Ejecutar();  
    if (MensajeRed())     ObtenMensaje();  
    if (TeclaPulsada())   ObtenTecla();  
    if (BloqueDiscoListo()) ObtenBloque();  
    ....  
}
```

- ! La producción del bucle está limitada por la función más lenta ; Esto **NO es aceptable en un SO real** (imagina: entorno de ventanas que no atiende al ratón mientras dibuja una ventana).



¿Por qué usar procesos ?

- **Simplicidad** – en un sistema existen muchas operaciones independientes (gcc, lpr, edit,..) que podemos envasar cada una en un proceso, acotando así sobre lo que tenemos que razonar.
- **Velocidad** – si un proceso se bloquea (esperando por disco, teclado, red,...) cambiamos a otro, como si tuviésemos más de una CPU.
- **Seguridad** – limitamos los efectos de un error.

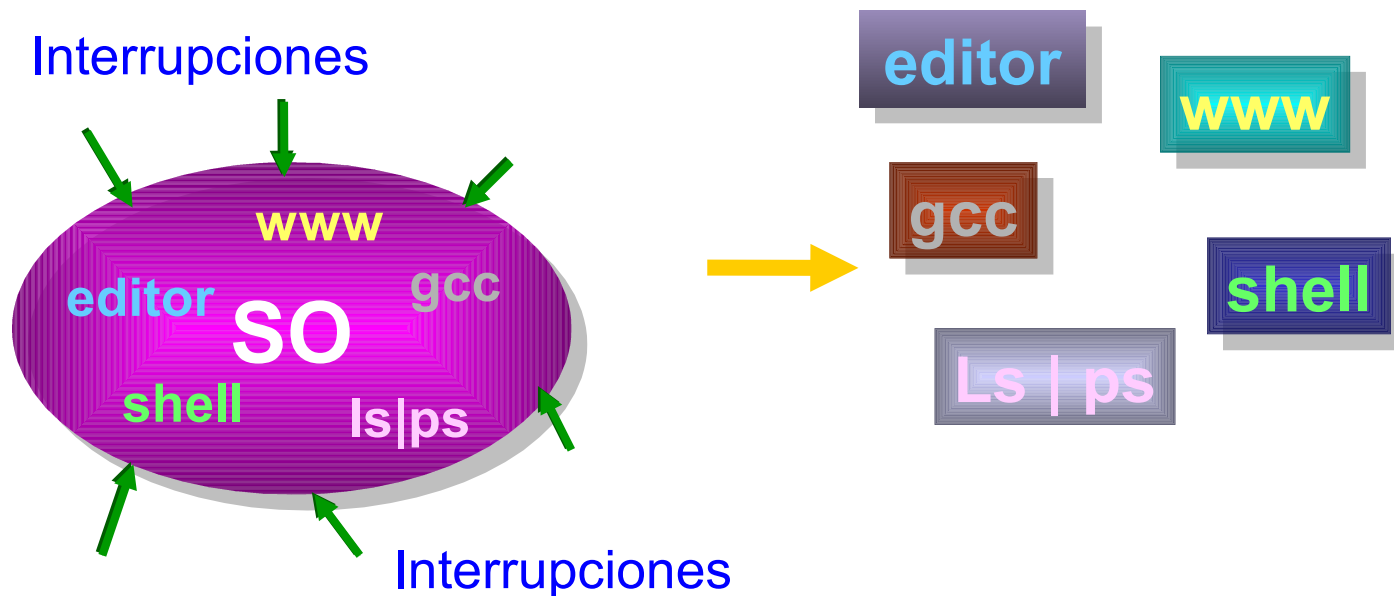


Usamos intercambiamente los términos proceso (*process*), tarea (*task*), y trabajo (*job*).



Simplicidad

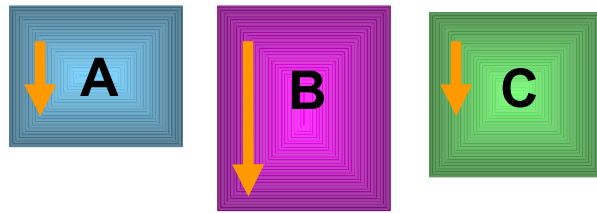
- Aislando cada actividad en un proceso:
 - tratamos con un proceso cada vez, y
 - los procesos sólo tratan con el SO.





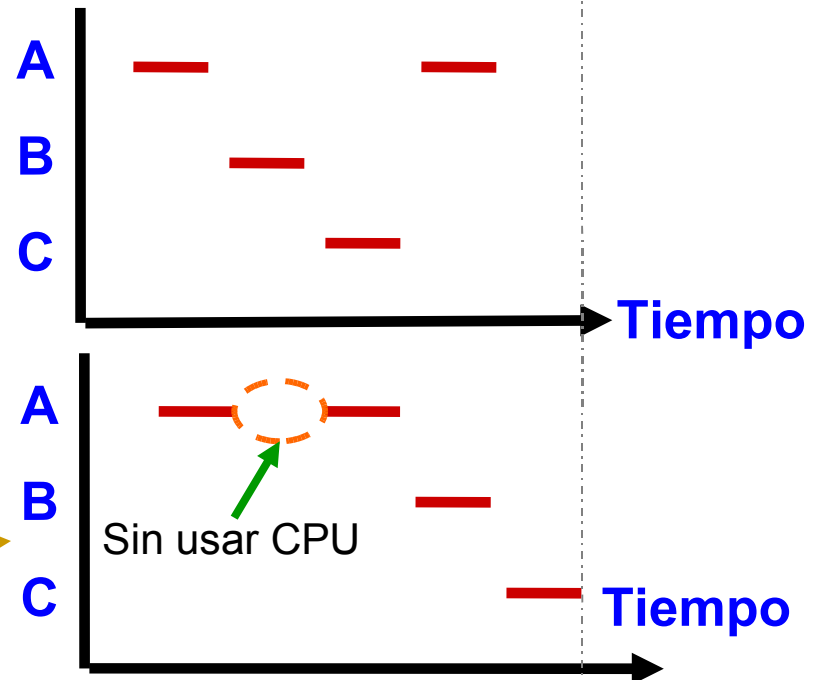
Velocidad

- Los SOs actuales permiten la ejecución simultánea de varios procesos (seudoparalelismo):



Vista conceptual

Ejecución Seudoparalela
Secuencial





Seguridad

- Vamos a hacer creer a los programas que están solos en la máquina.
- Cada proceso se ejecuta en **SU propio espacio de direcciones** -> no puede acceder directamente a espacios de direcciones de otros procesos (lo veremos en los Temas 4-5).
- Su ejecución esta confinada a su espacio y también sus errores.
- **Coste**: compartir información se complica

En la vida real ...



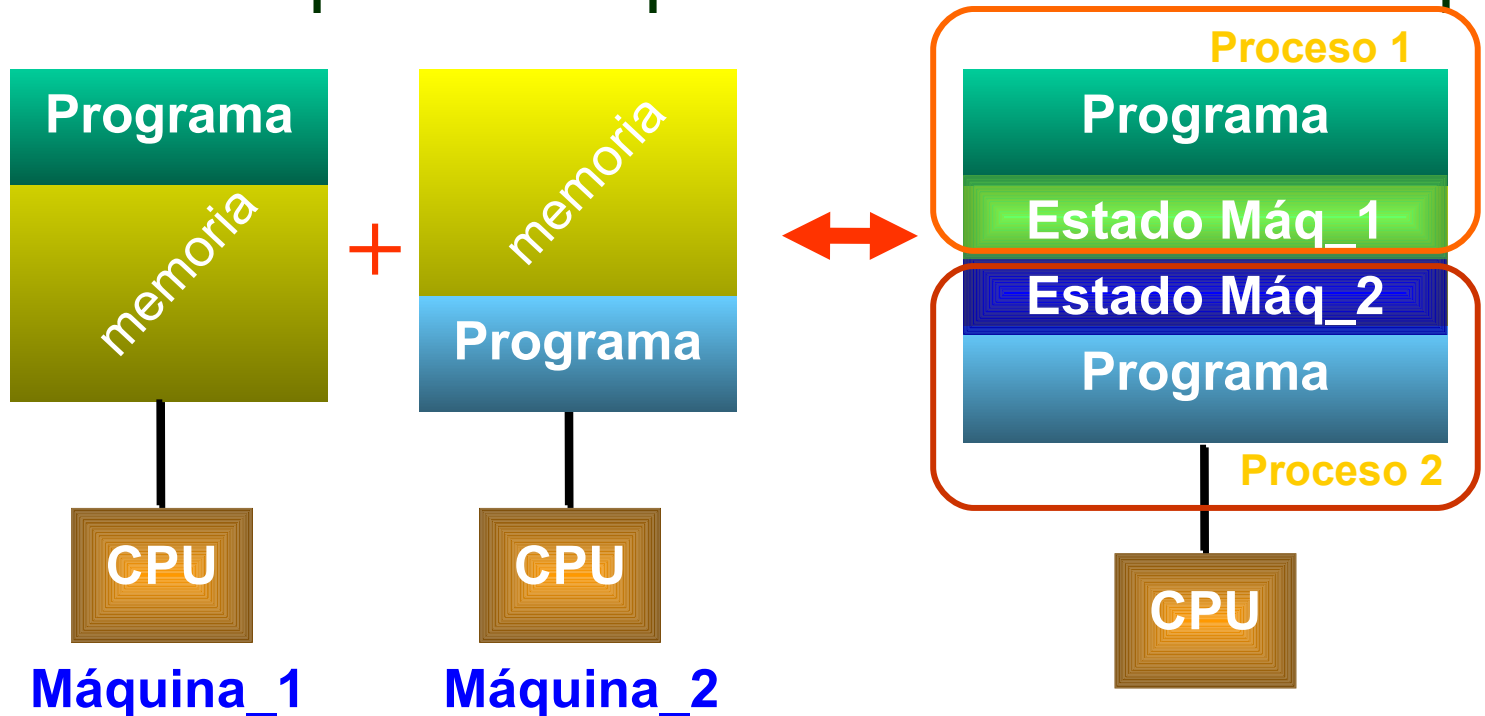


Proceso = CPU virtual

- **Proceso** = “programa en ejecución” = un flujo secuencial de ejecución en su propio espacio de direcciones.
- Un proceso es un recurso virtual – una abstracción que desacopla la CPU física del recurso de computo presentado al programa – creamos la ilusión de tener más de una CPU.

Idea de implementación

- Multiplexamos la CPU en el tiempo, hacemos creer a cada proceso que es único en la máq.





¿Qué hay en un proceso?

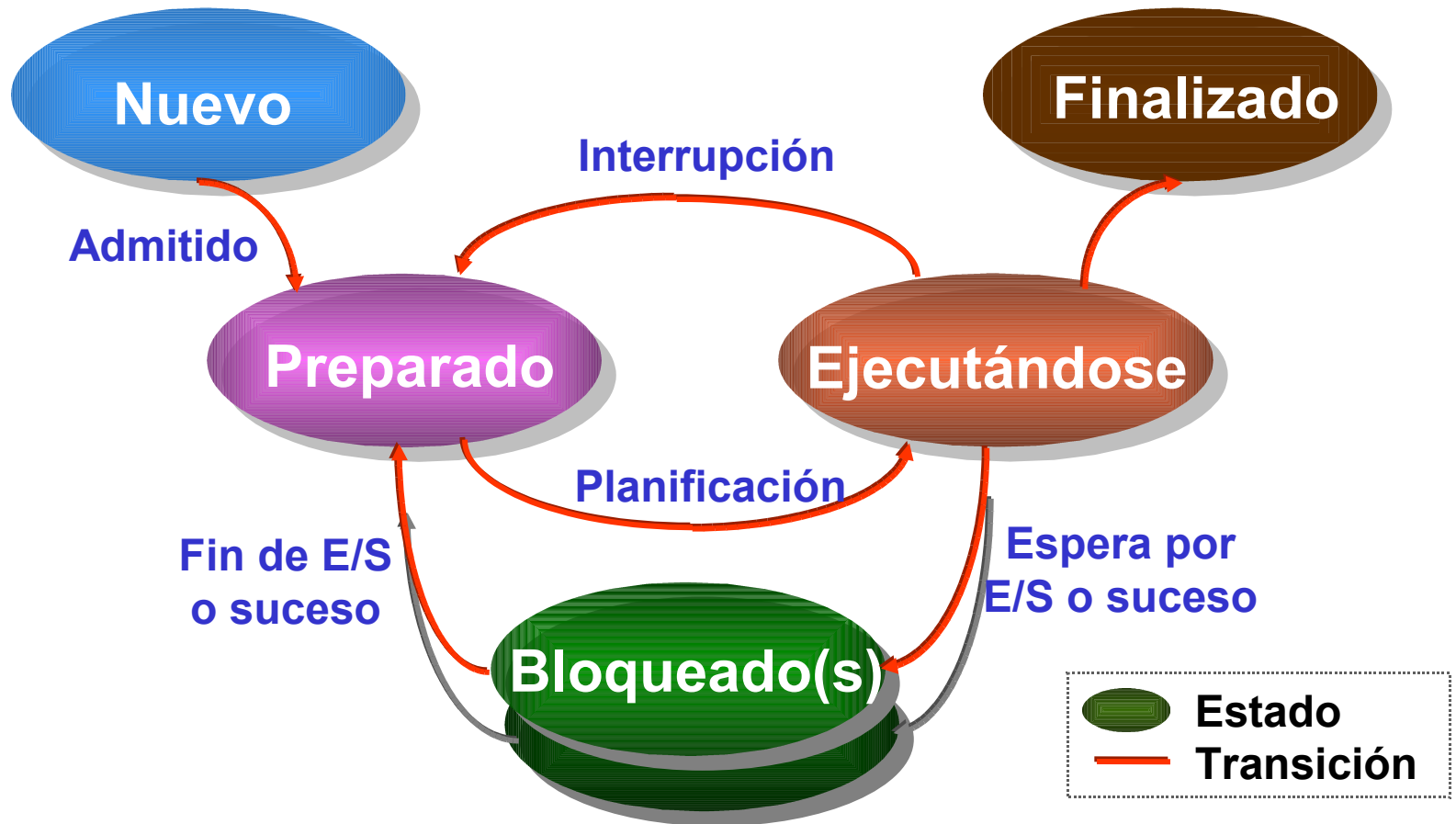
- Para representar un proceso debemos recoger toda la información que nos de el estado de ejecución de un programa:
 - el código y datos del programa.
 - una pila de ejecución.
 - el PC indicando la próxima instrucción.
 - los valores actuales del conjunto de registros de la CPU.
 - un conjunto de recursos del sistema (memoria, archivos abiertos, etc.).



Estado de un proceso

- Varios procesos pueden ejecutar incluso el mismo programa (p.ej. un editor) pero cada uno uno tiene su propia representación.
- Cada proceso está en un **estado de ejecución** que caracteriza lo que hace y determina las acciones que se pueden sobre él.
 - **Preparado** – en espera de la CPU.
 - **Ejecutándose** – ejecutando instrucciones.
 - **Bloqueado** – esperando por un suceso.
- Conforme un programa se ejecuta, pasa de un estado a otro.

Diagrama de estados





Bloque de Control de Proceso

- **Bloque de Control del Proceso (PCB)** – estructura de datos que representa al proceso; contiene su información asociada:
 - Identificador del proceso en el sistema.
 - Estado del proceso.
 - Copia de los registros de la CPU.
 - Información de planificación.
 - Información para la gestión de memoria.
 - Información del estado de las E/S.
 - Información de contabilidad. ...



PCB's y estado hardware

- Cuando un proceso esta ejecutándose, los valores de su PC, puntero a pila, registros, etc., es decir, su **contexto**, está cargado en los registros de la CPU.
- Cuando el SO detiene la ejecución de un proceso, salva su contexto en su PCB.
- La acción de conmutar la CPU de un proceso a otro se denomina **cambio de contexto**. Los sistemas de tiempo compartido realizan de 10 a 100 cambios de contexto por segundo.
- Un cambio de contexto suele tardar menos de 1 ms. Este trabajo es puramente sobrecarga.



Contexto y modo dual

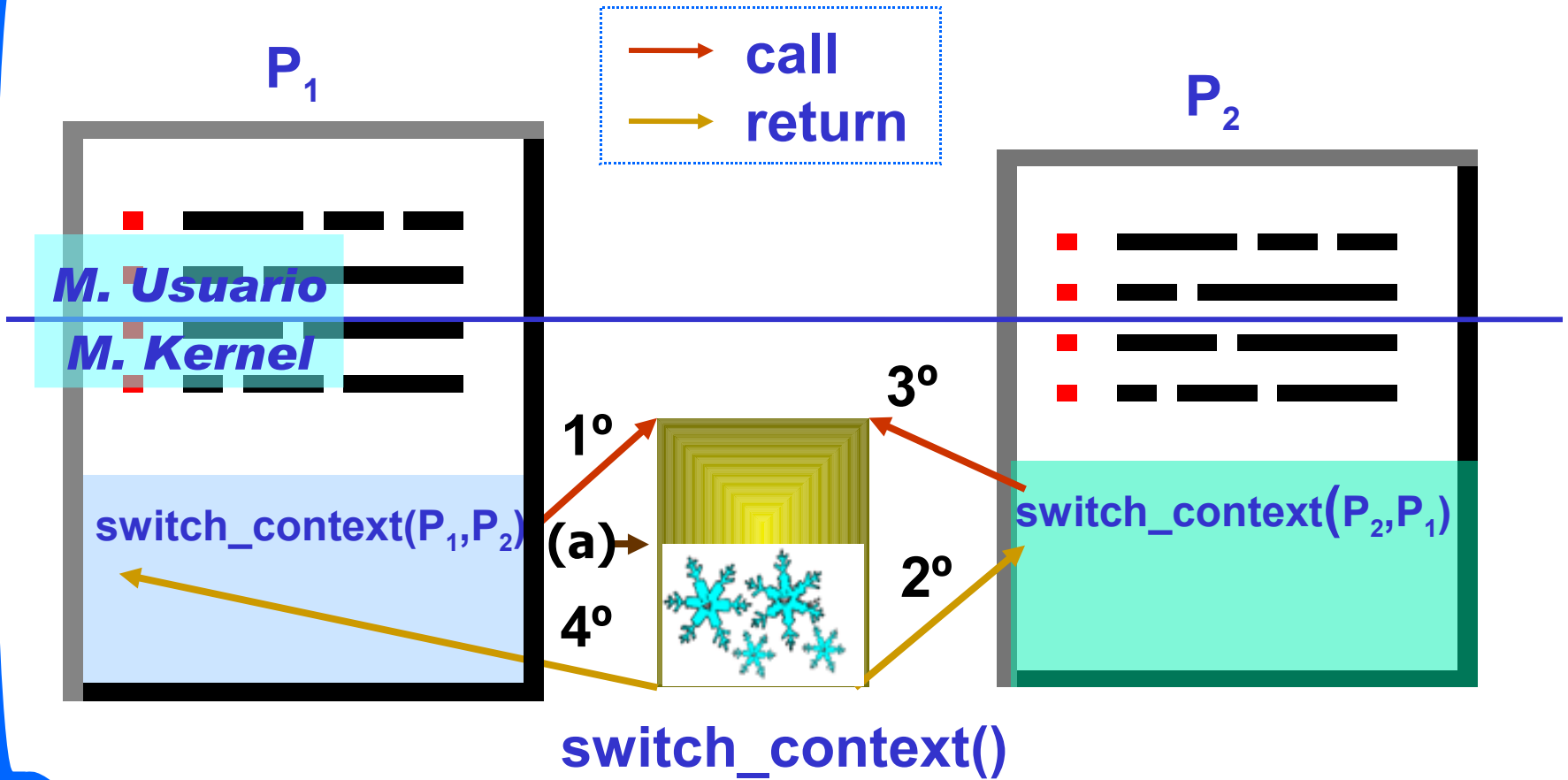
- En máquinas con dos modos de funcionamiento, separamos el contexto en dos, uno por modo:
 - **Contexto de usuario** – contexto utilizado cuando se ejecuta en modo usuario.
 - **Contexto en modo kernel** – cuando el proceso se ejecuta en modo kernel.
- Al pasar un proceso de modo usuario a kernel, el SO salva el contexto de usuario y utiliza el de modo kernel. Al retornar, restauramos el contexto de usuario.



Cambio de contexto (i)

- Cuando hacemos un cambio de contexto, debemos:
 - Salvar el contexto previo.
 - Restaurar el contexto nuevo.
- P. ej. `switch_context(P1, P2)` podría ser una rutina en lo más profundo del SO que salva el contexto de P_1 en su PCB, y restaura el nuevo contexto del PCB de P_2 .
- ¿En qué se diferencia ésta función de una llamada a procedimiento normal? ...

Cambio de contexto (ii)



... en detalle a continuación



Ilustración del cambio de contexto: supuestos

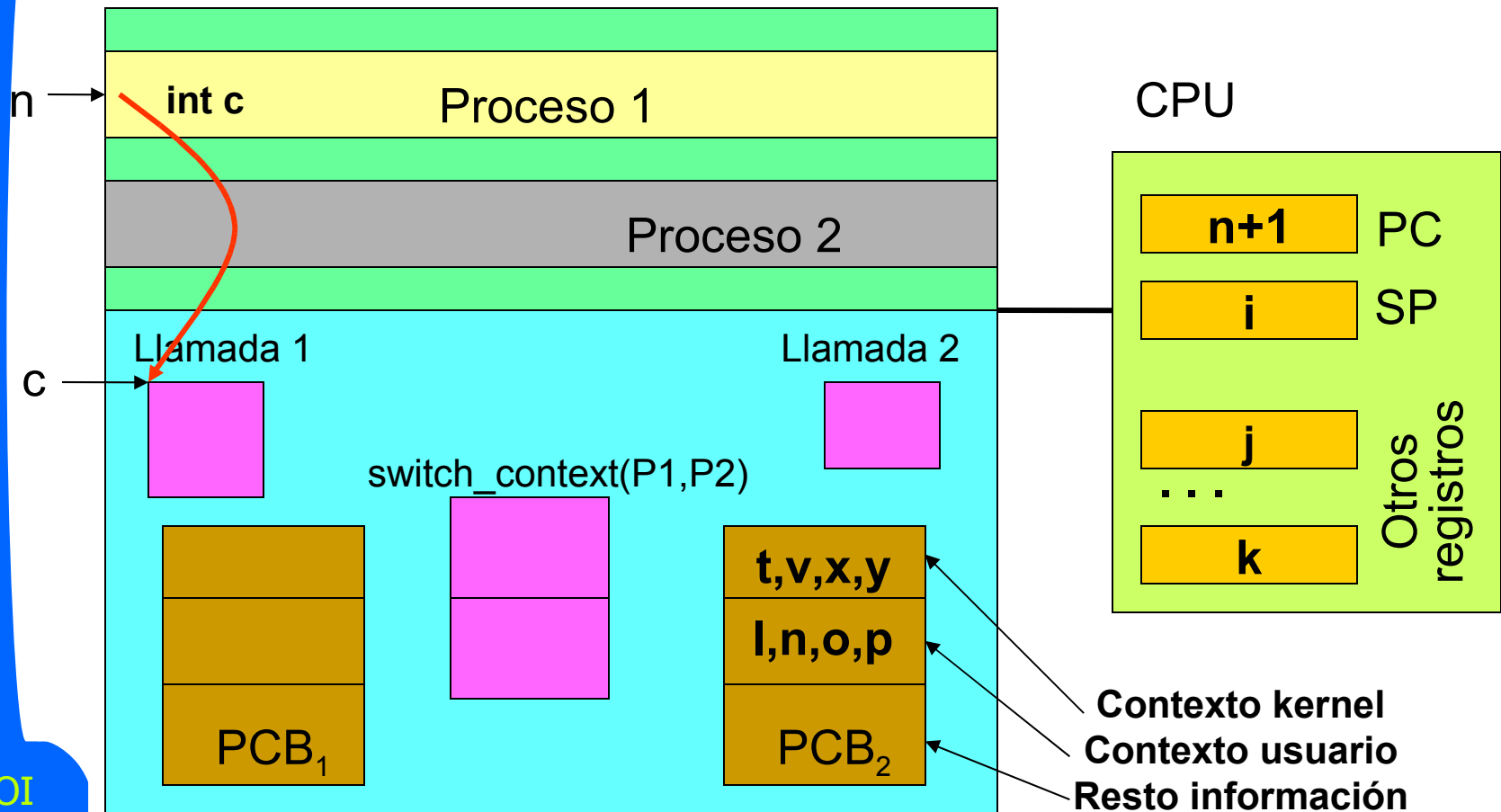
- Suponemos dos procesos:
 - P1 esta ejecutando la instrucción n que es una llamada al sistema.
 - P2 se ejecutó anteriormente y ahora esta en el estado preparado esperando su turno.
- Convenio:
 - Código del SO
 - Estructura de datos
 - Flujo de control
 - - -> Salvar estructuras de datos
 - Instrucción i-ésima a ejecutar



1º - P_1 ejecuta n

Memoria

Máquina en modo usuario

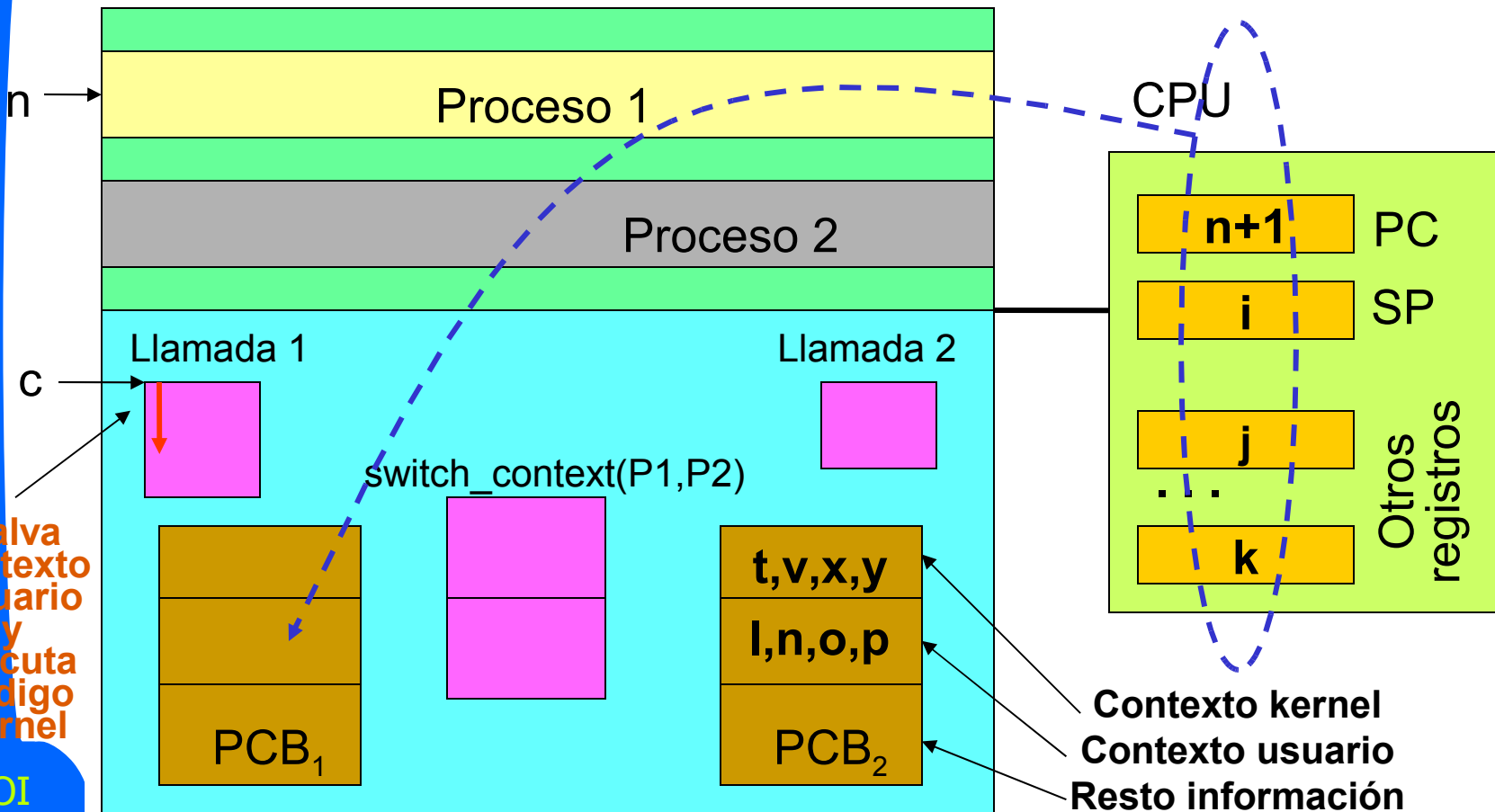




2º - Salva contexto usuario y ejecuta f^{on} kernel

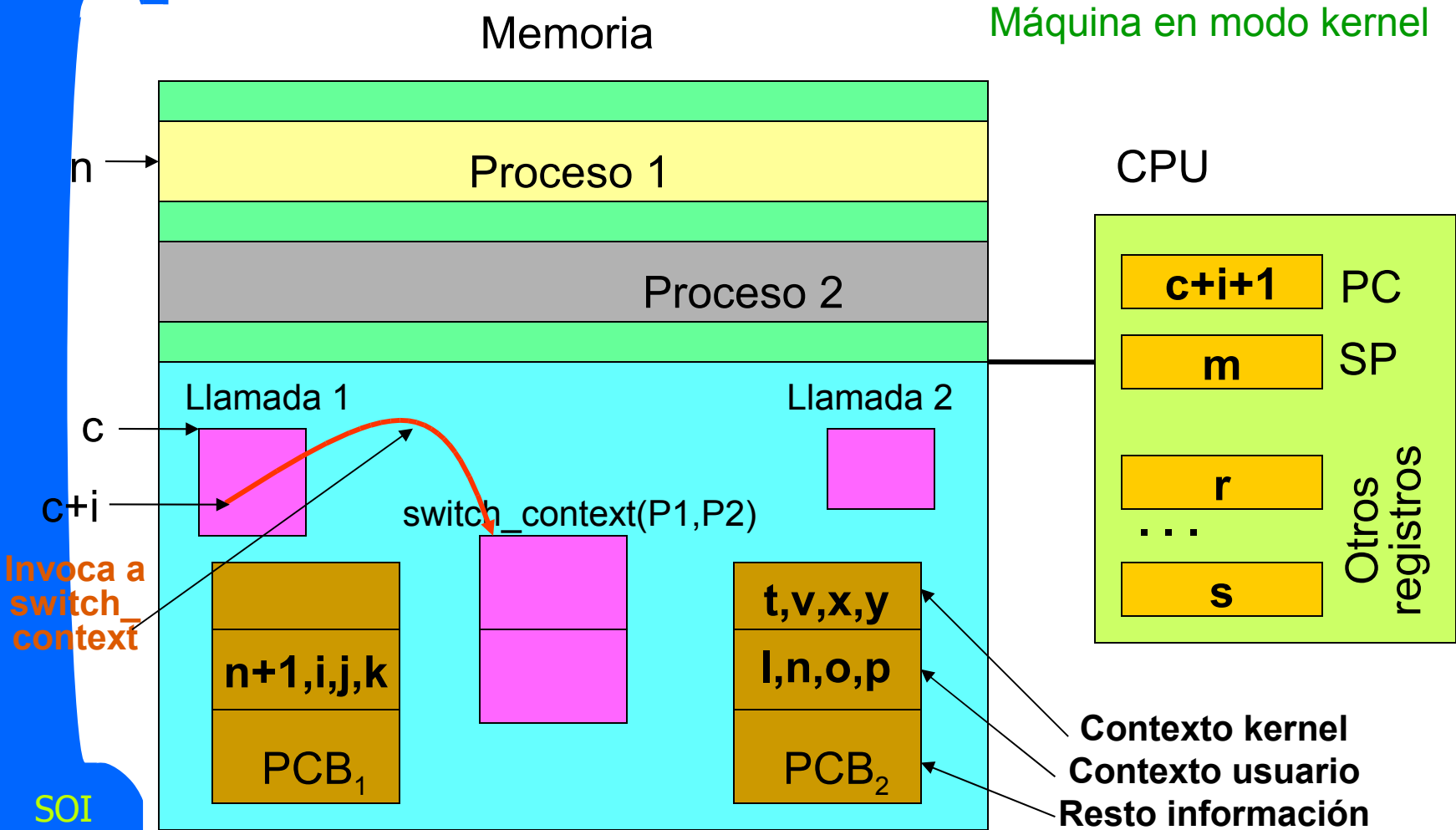
Memoria

Máquina en modo kernel





3º - Parar proceso, invoca a cambio_contexto

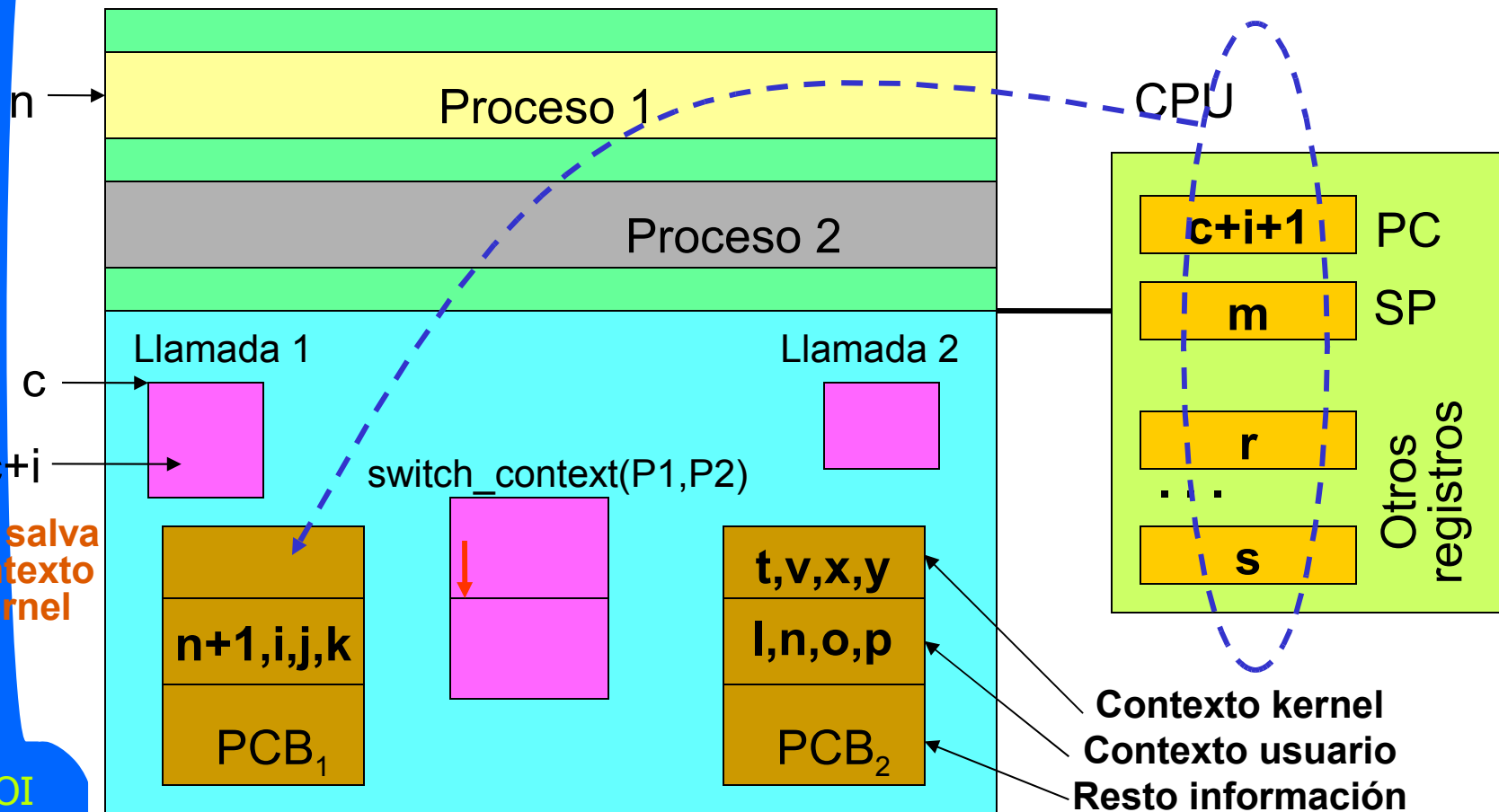




4º - Cambio_contexto () salva contexto kernel

Memoria

Máquina en modo kernel



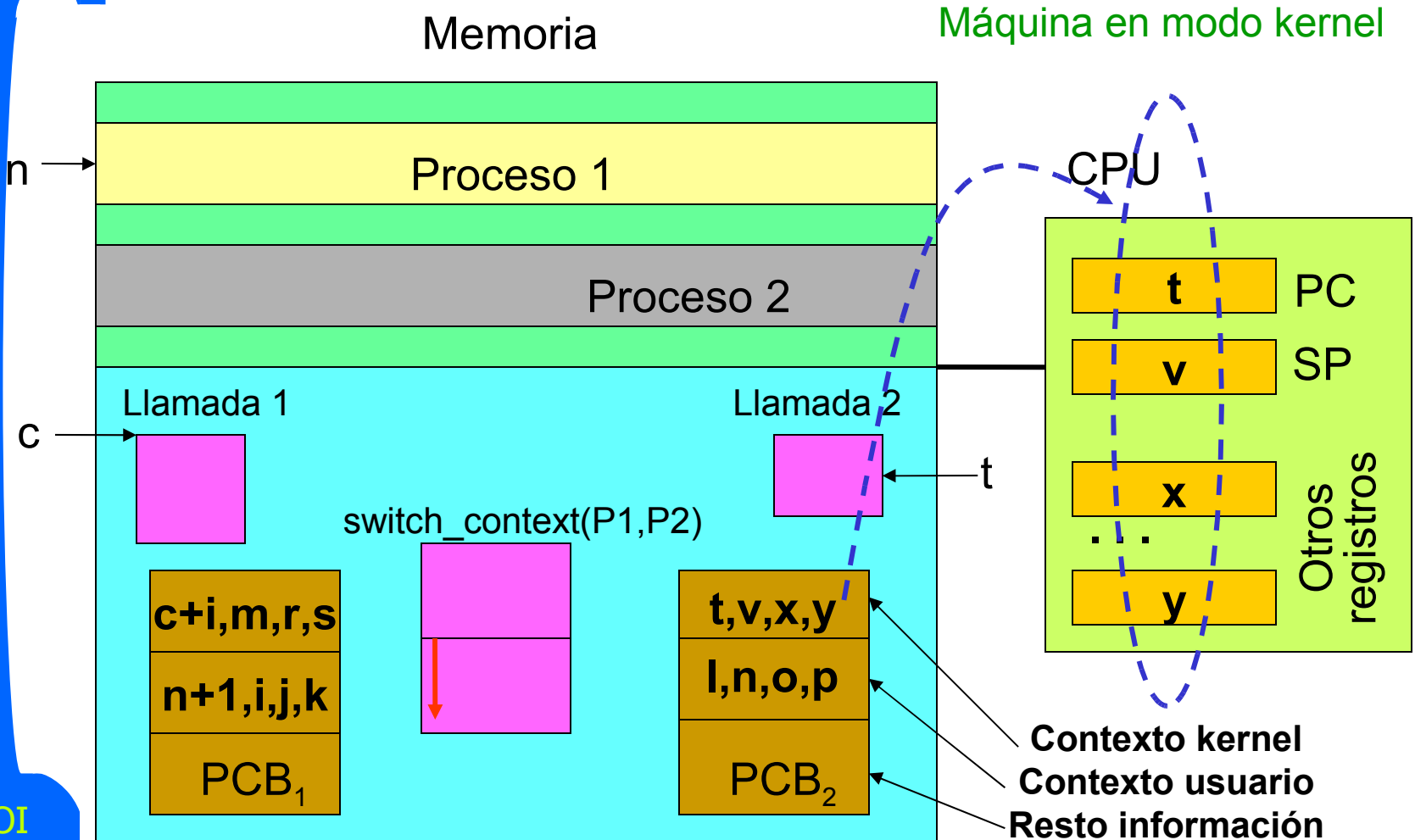


¿Cómo estamos?

- Llegados a este punto P_1 esta detenido, “congelado” y nos disponemos a reanudar, “descongelar”, a P_2 (que previamente habíamos parado en algún instante anterior).
- Es decir, estamos en el punto marcado como (a) en la transparencia 18.



5^o - Repone contexto kernel de P₂

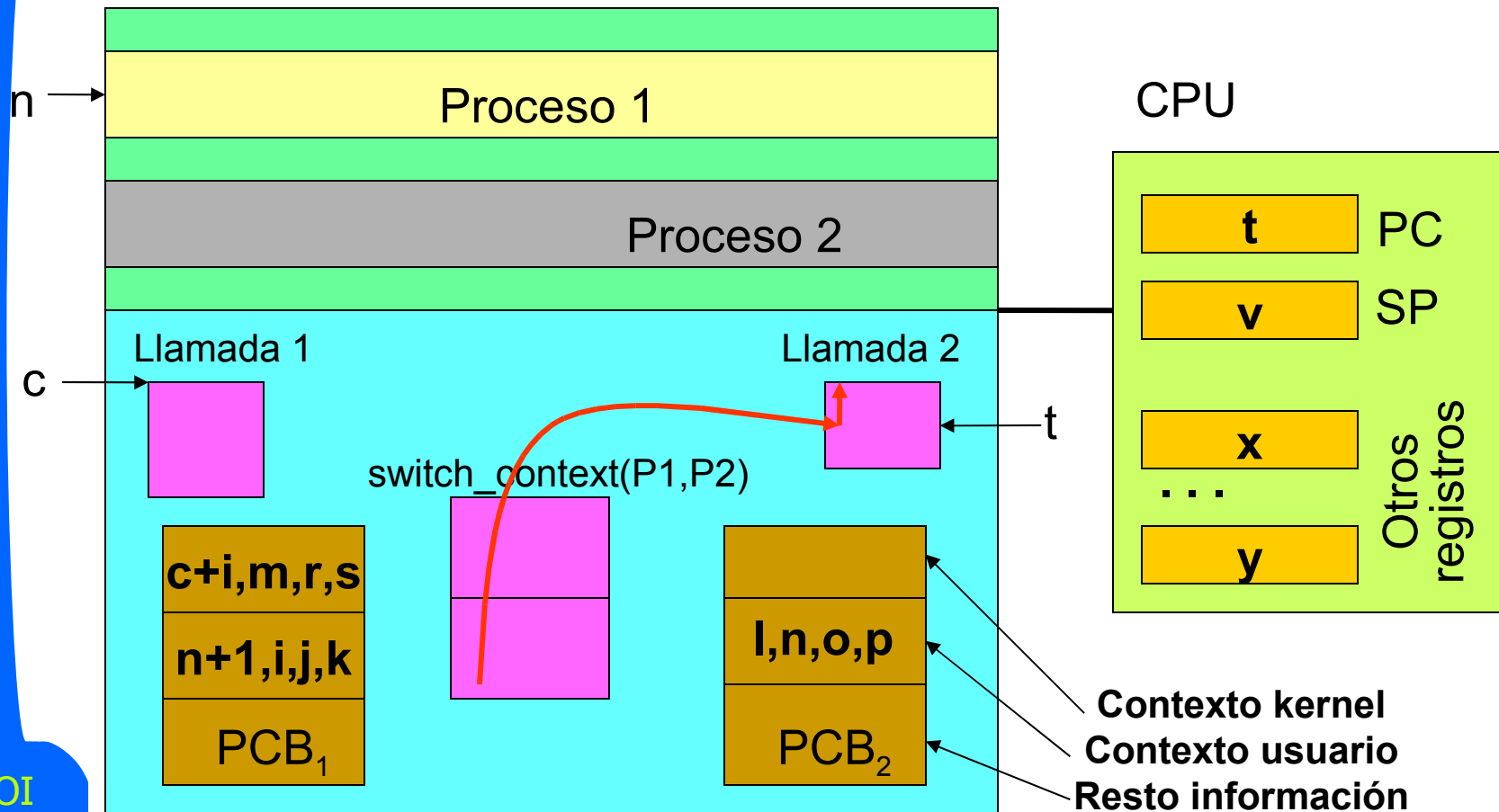




6^o - El kernel termina la fon que inicio de P₂

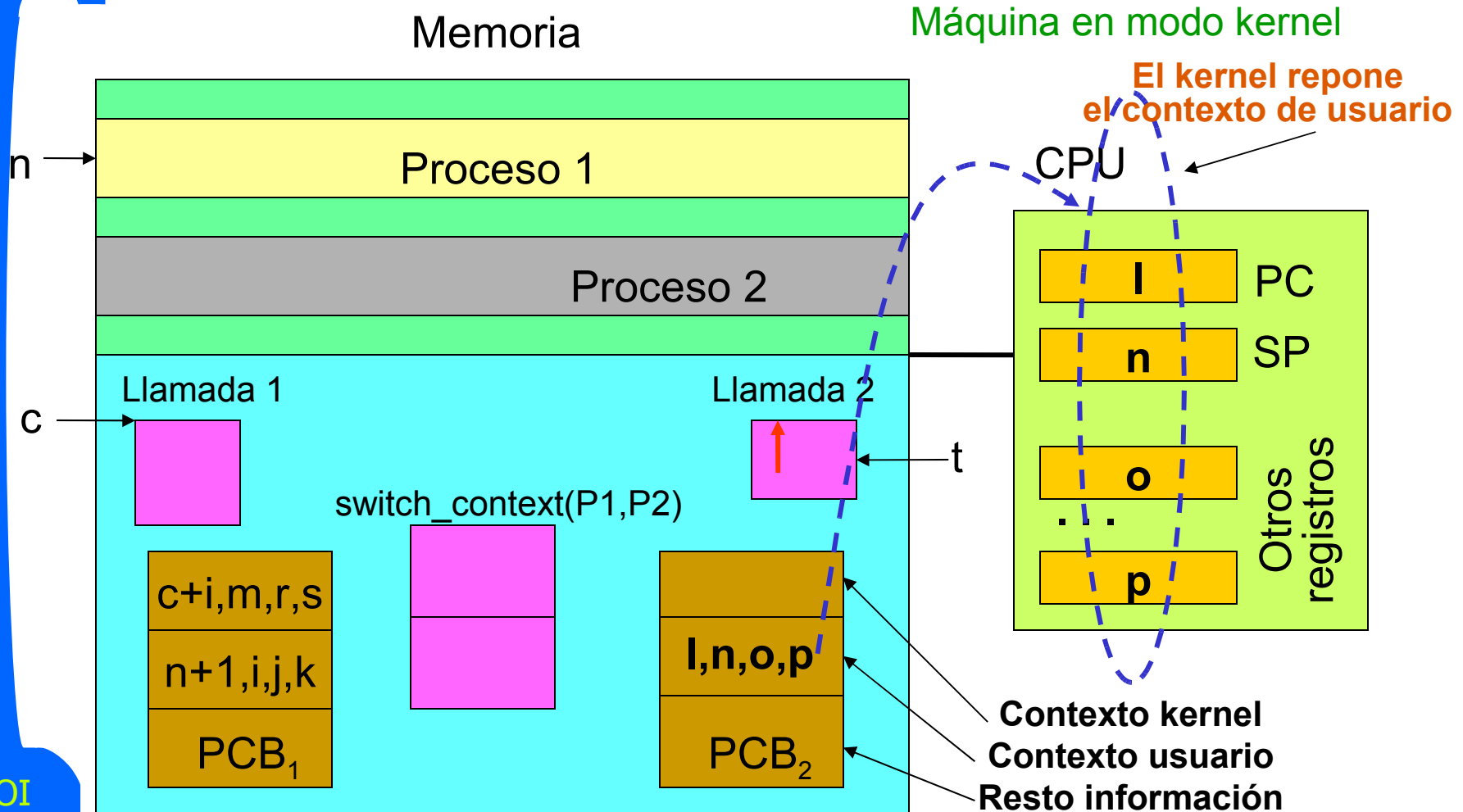
Memoria

Máquina en modo kernel





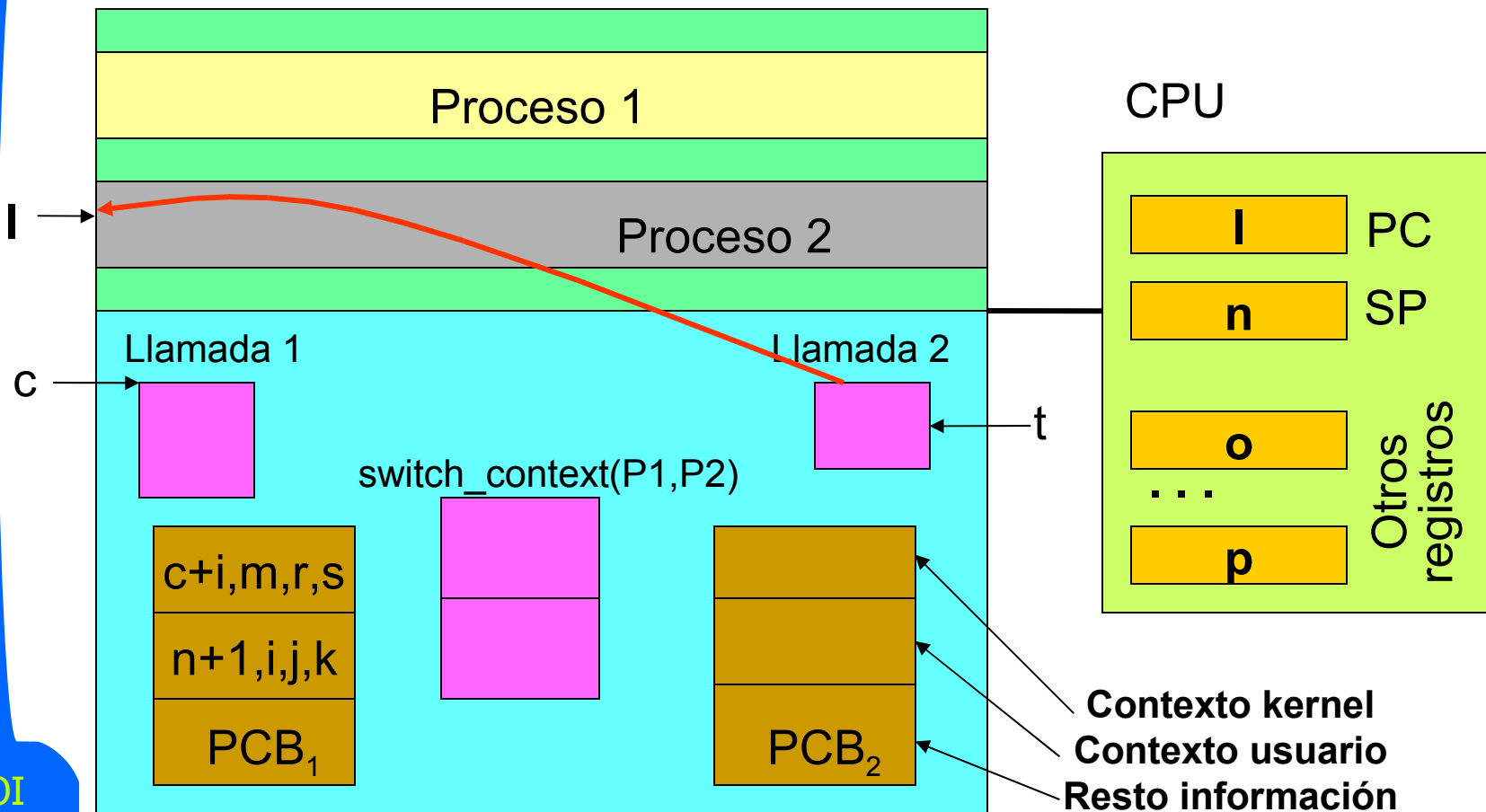
7º - Finalizada f^{on}, retorna a modo usuario





8º - reanudamos ejecución de P₂

Máquina en modo usuario





Observaciones

- Cuando conmutamos al proceso P_2 , este tiene la estructura de PCB que aparece en el dibujo adjunto. Es decir, hemos supuesto que se ha ejecutado con anterioridad.
- ¿Qué pasa si acabo de lanzar P_2 ?





Respuesta

- La llamada al sistema *crear_proceso()* esta diseñada para crear un proceso cuyo PCB tiene la estructura anterior.
- ¿Qué valores tiene el contexto de este PCB?
 - El SO ajusta los valores del contexto de usuario para que el proceso recién creado se ejecute desde su primera instrucción (PC = 1ª instrucción).
 - Se crea un contexto kernel para que parezca que el proceso retorna de una llamada al sistema.
- Nueva pregunta ¿por qué hacer esto así? ...
Respuesta en Problema 4 de Temas 1 y 2.



Mecánica del cambio de contexto

- El cambio de contexto es muy dependiente de la máquina: salvar registros generales y de punto flotante, estado del coprocesador, ...
- El coste del cambio de contexto proviene:
 - Coste directo de salvar registros de propósito general y los especiales.
 - Coste indirecto de limpieza de cachés (los datos de las cachés del proceso actual son inválidos para el entrante).



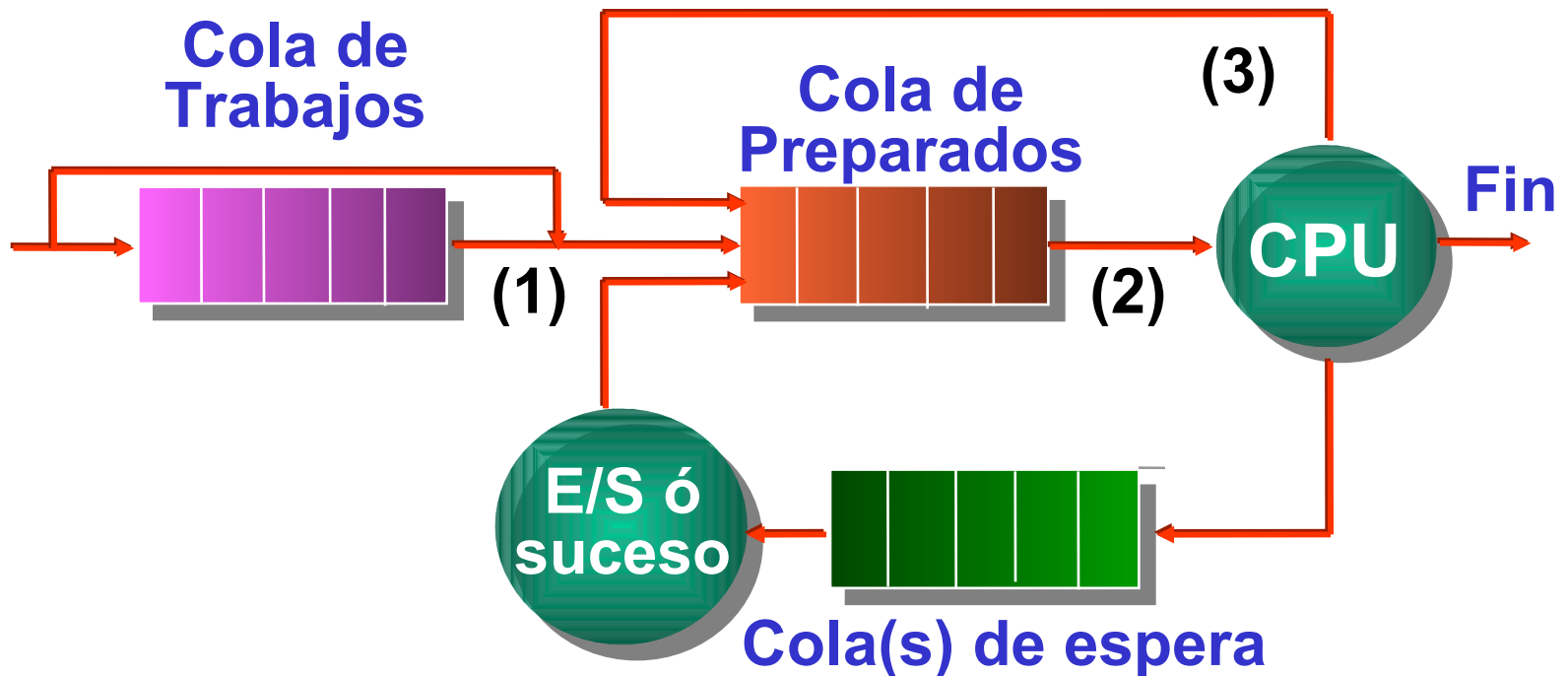
PCBs y Colas de Estados

- El SO mantiene una cola de PCBs por estado; cada una de las cuales contiene a los procesos que están en ese estado.
- De esta forma:
 - Al crear un proceso, su PCB encola en la cola de estado acorde a su estado actual.
 - Conforme un proceso cambia de estado, su PCB es retirado de una cola y encolado en otra.



Modelo de sistema

- Podemos modelar el sistema como un conjunto de procesadores y de colas:





Colas de estados

- **Cola de trabajos** – conjunto de todos los procesos del sistema.
- **Cola de preparados** – conjunto de todos los procesos que residen en memoria principal, preparados y esperando para ejecutarse.
- **Cola(s) de espera** – conjunto de todos los procesos esperando por un dispositivo de E/S particular o por un suceso (búfer de memoria, un semáforo, etc.).



Hebras (o hilos)

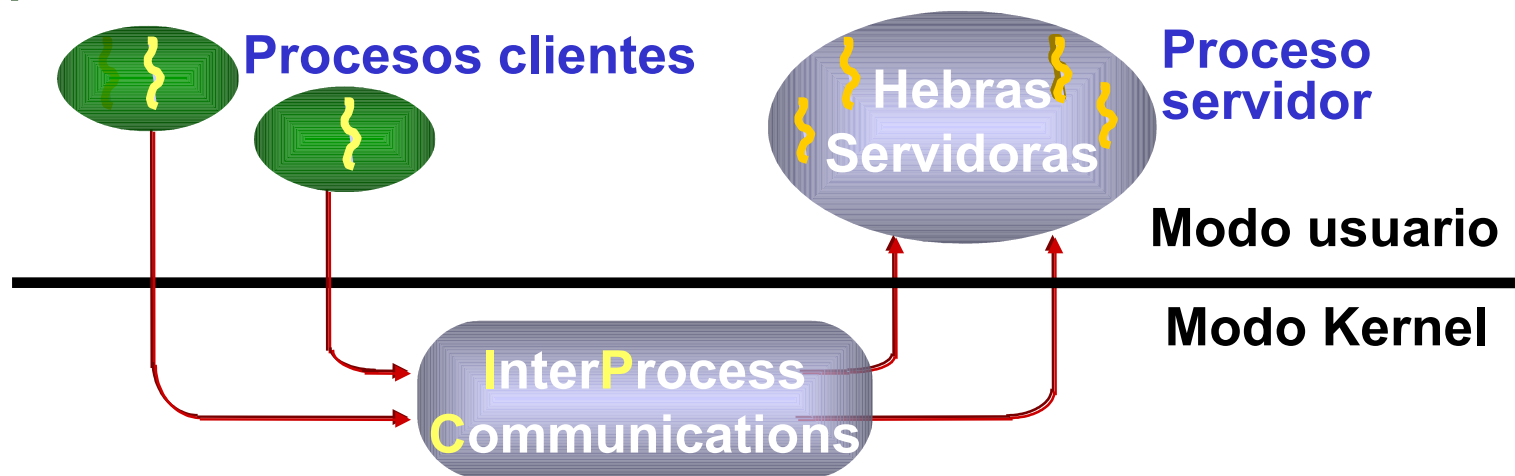



- Concepto
- Tipos:
 - Usuario
 - Kernel
 - Procesos ligeros.
- Hebra en Solaris

Limitaciones de los procesos



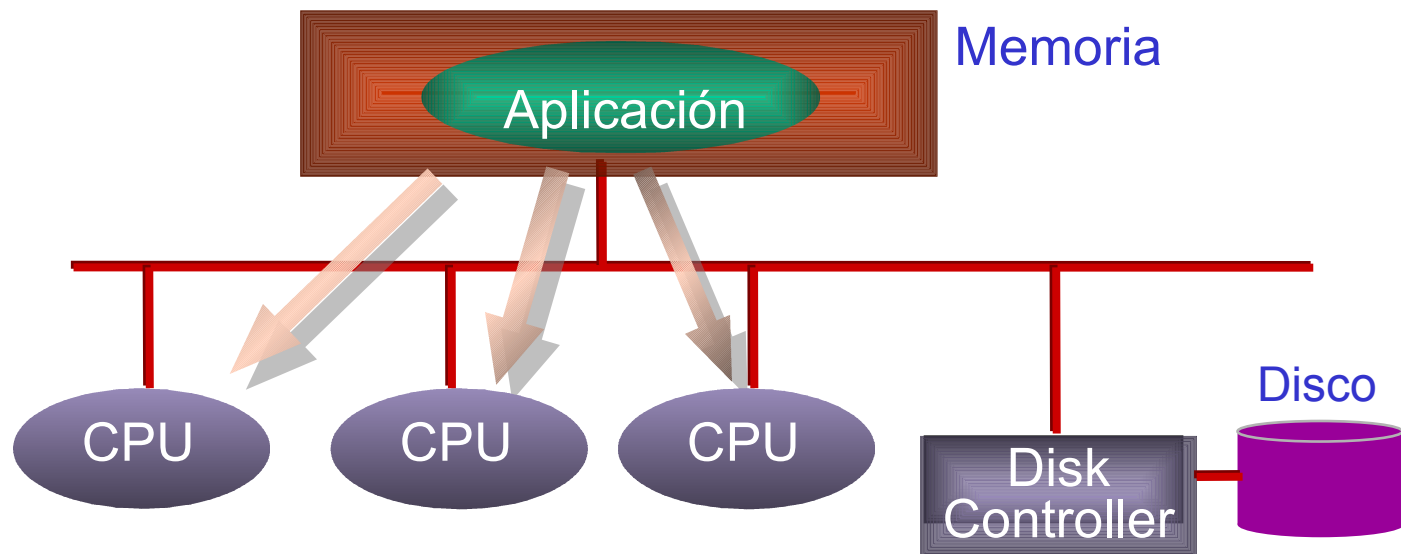
- Hay aplicaciones que siendo por naturaleza paralelas necesitan un espacio común de direcciones. Ejemplos: servidores de bases de datos, monitores de procesamiento de transacciones, procesamiento de protocolos de red, etc.





Limitaciones de los procesos(ii)

1. Los procesos no pueden sacar partido de las arquitecturas multi-procesadoras dado que un proceso sólo puede usar un procesador a la vez.





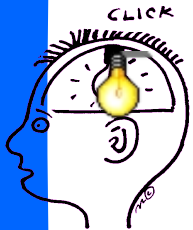
Concepto proceso revisado

- En un proceso confluyen dos ideas que podemos separar:
 - **flujo de control** – secuencia de instrucciones a ejecutar; determinadas por el PC, la pila y los registros.
 - **espacio de direcciones** – direcciones de memoria a las que puede acceder y recursos asignados (archivos, memoria, etc.).



Hebras ó hilos (*threads*)

- Queremos introducir un modelo de ejecución diferente del modelo de proceso visto, pero queremos que sea compatible.

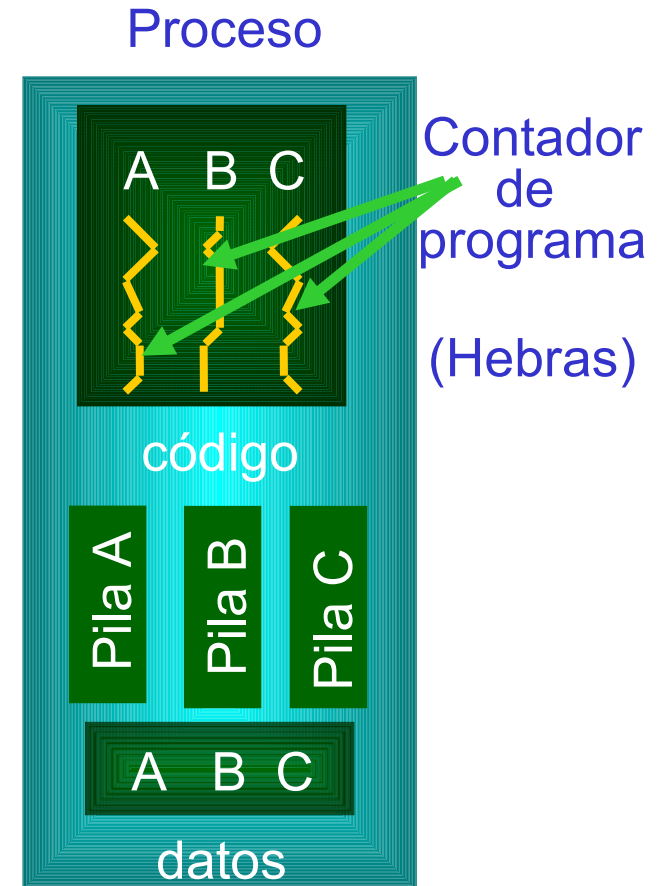


Permitir más de un flujo de control dentro del mismo espacio de direcciones – dentro del mismo programa (datos y código) y los mismos recursos del SOs, permitimos varias ejecuciones.



Definición

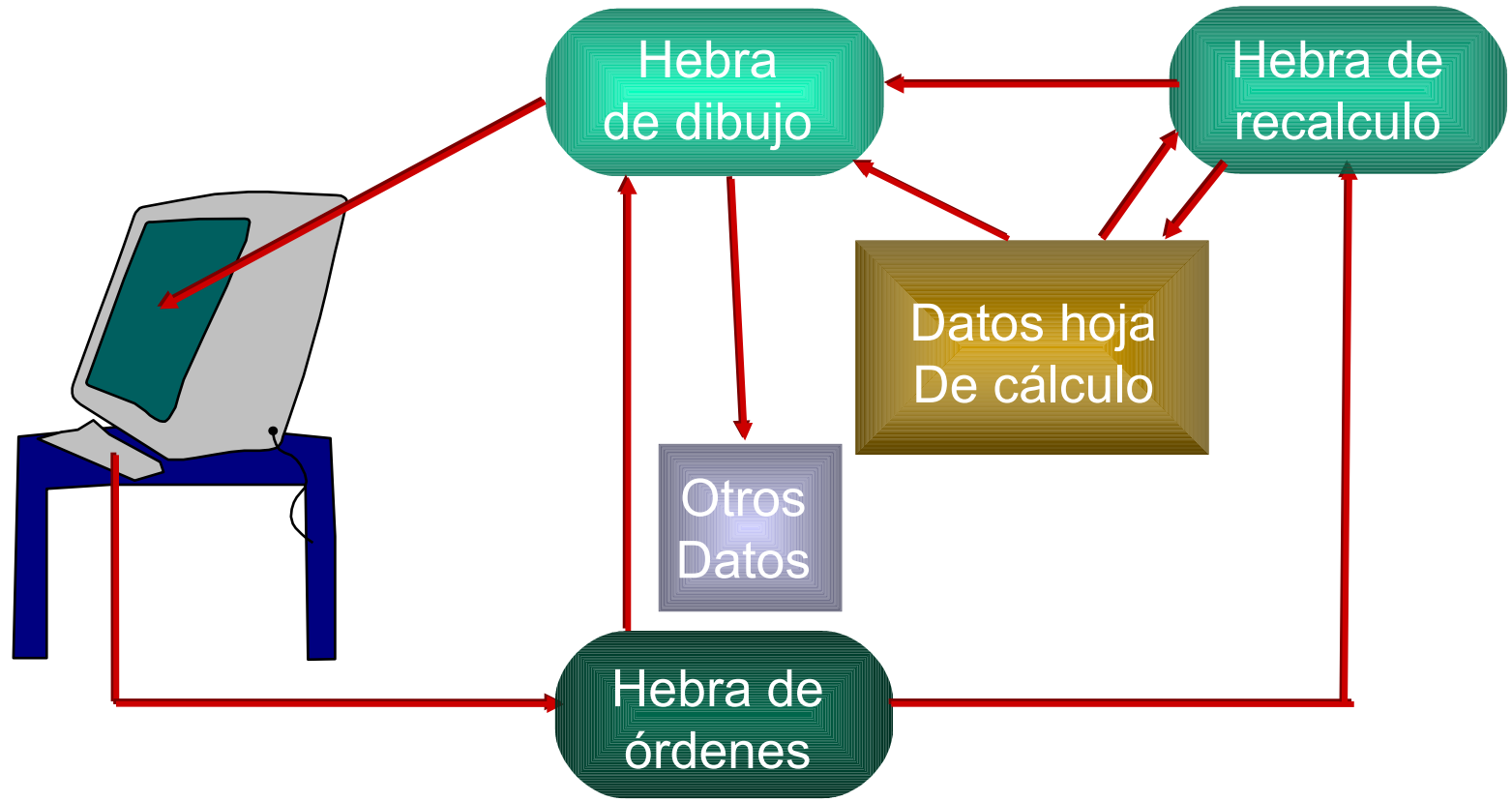
- **Hebra** = unidad de asignación de la CPU (de planificación).
- Tiene su propio contexto hardware:
 - Su valor del contador de programa.
 - Los valores de los registros.
 - Su pila de ejecución.



“ Programa multihebrado ”



Hoja de cálculo multihebrada





¿Qué programas multihebrar?

- Tareas independientes
 - Ej.: depurador necesita GUI, ...
 - E/S asíncronas.
- Programas únicos, operaciones concurrentes:
 - Servidores: de archivos, web, etc.
 - Kernels de SOs: ejecución simultánea de varias peticiones de usuario; no se necesita protección.
- Uso del hardware multiprocesador.



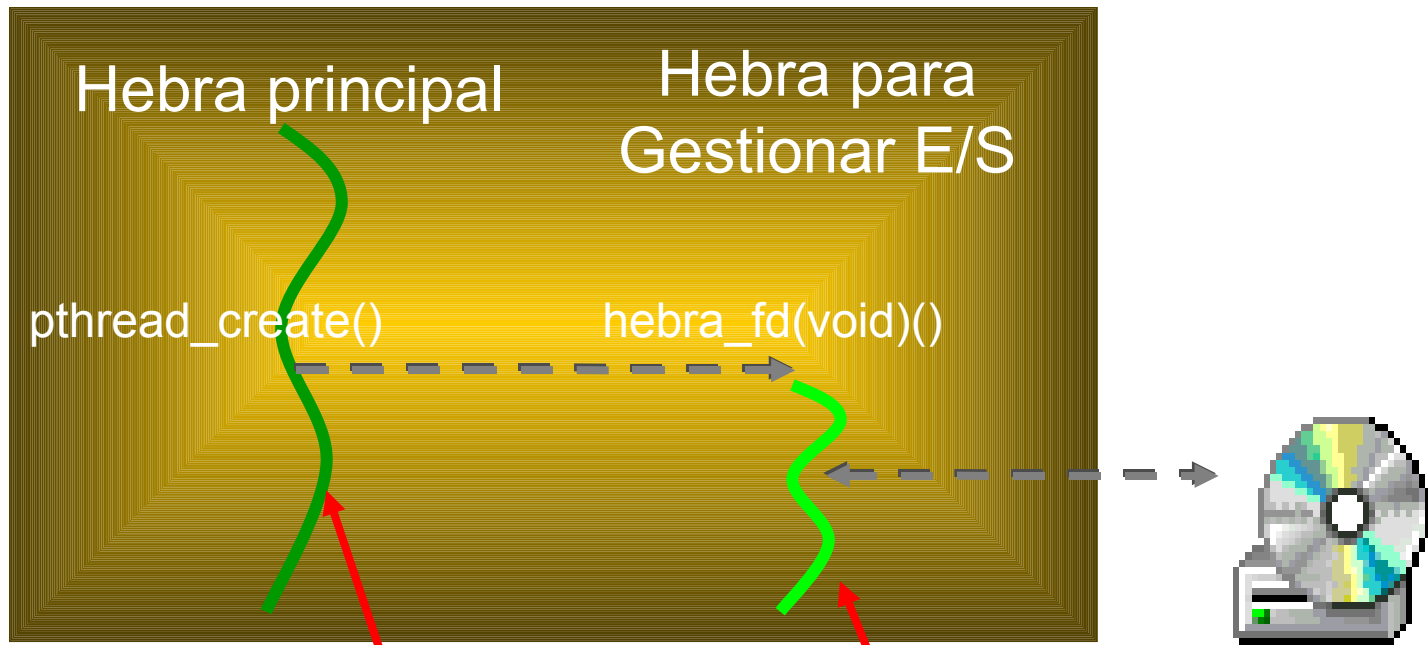
¿Qué no multihebrar?

- Si todas las operaciones son CPU intensivas, no llegaremos muy lejos multihebrando.
- La creación de hebras es barata, pero no libre; una hebra con 5 líneas de código no es muy **útil**.



E/S asíncronas

Proceso multihebrado



E/S asíncrona para la hebra principal
E/S síncrona para la hebra que gestiona la E/S



Ejemplo:

- Ilustraremos la utilidad de las E/S asíncronas, para ello, vamos a dibujar un **Conjunto de Mandelbrot** (un fractal) en tres versiones:
 - **Secuencial** – una sola hebra tanto para dibujar la ventana como para realizar el dibujo.
 - **Multihebrada** – una hebra para dibujar la ventana, y otra para el dibujo.
 - **Multihebrada con dos hebras** para dibujo.

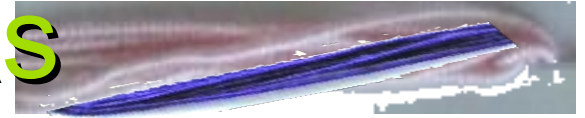


Estándares de hebras

- POSIX (Pthreads) – ISO/IEEE estándar
 - API común
 - Casi todos los UNIX tienen una biblioteca de hebras.
- Win32
 - Muy diferente a POSIX
 - Existen bibliotecas comerciales de POSIX
- Solaris
 - Anterior a POSIX; probablemente lo será.
- Hebras Java



Tipos de hebras



- **Hebras Kernel** – implementadas dentro del kernel. Conmutación entre hebras rápida.
- **Hebras de usuario** – implementadas a través de una biblioteca de usuario que actúa como un kernel miniatura. La conmutación entre ellas es muy rápida.
- **Enfoques híbridos** – implementan hebras kernel y de usuario y procesos ligeros (p.ej. Solaris 2).



Hebras de usuario

- Alto rendimiento al no consumir recursos kernel (no hacen llamadas al sistema).
- El tamaño crítico de estas es del orden de unos cientos de instrucciones.
- Al no conocer el kernel su existencia ⇒
 - No aplica protección entre ellas.
 - Problemas de coordinación entre el planificador de la biblioteca y el del SO.
 - Si una hebra se bloquea, bloquea a la tarea completa.



Solaris 2.x

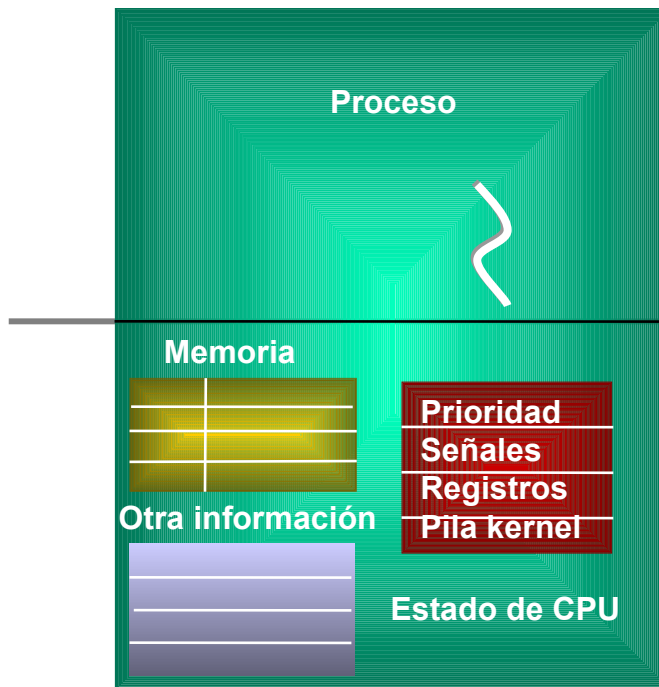


- **Solaris 2.x** es una versión multihebrada de Unix, soporta multiprocesamiento simétrico y planificación en tiempo real.
- **Lightweight Process (LWP)** – “hebra de usuario soportada por el kernel”. La conmutación entre LWP’s es relativamente lenta pues involucra llamadas al sistema. El kernel sólo ve los LWPs de los procesos de usuario.

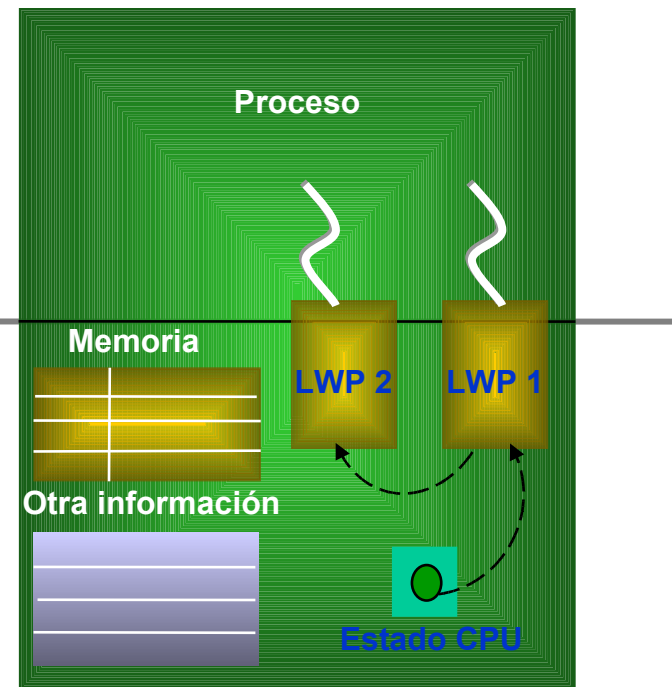


Implementación de LWP

Estructura de un proceso
Unix tradicional

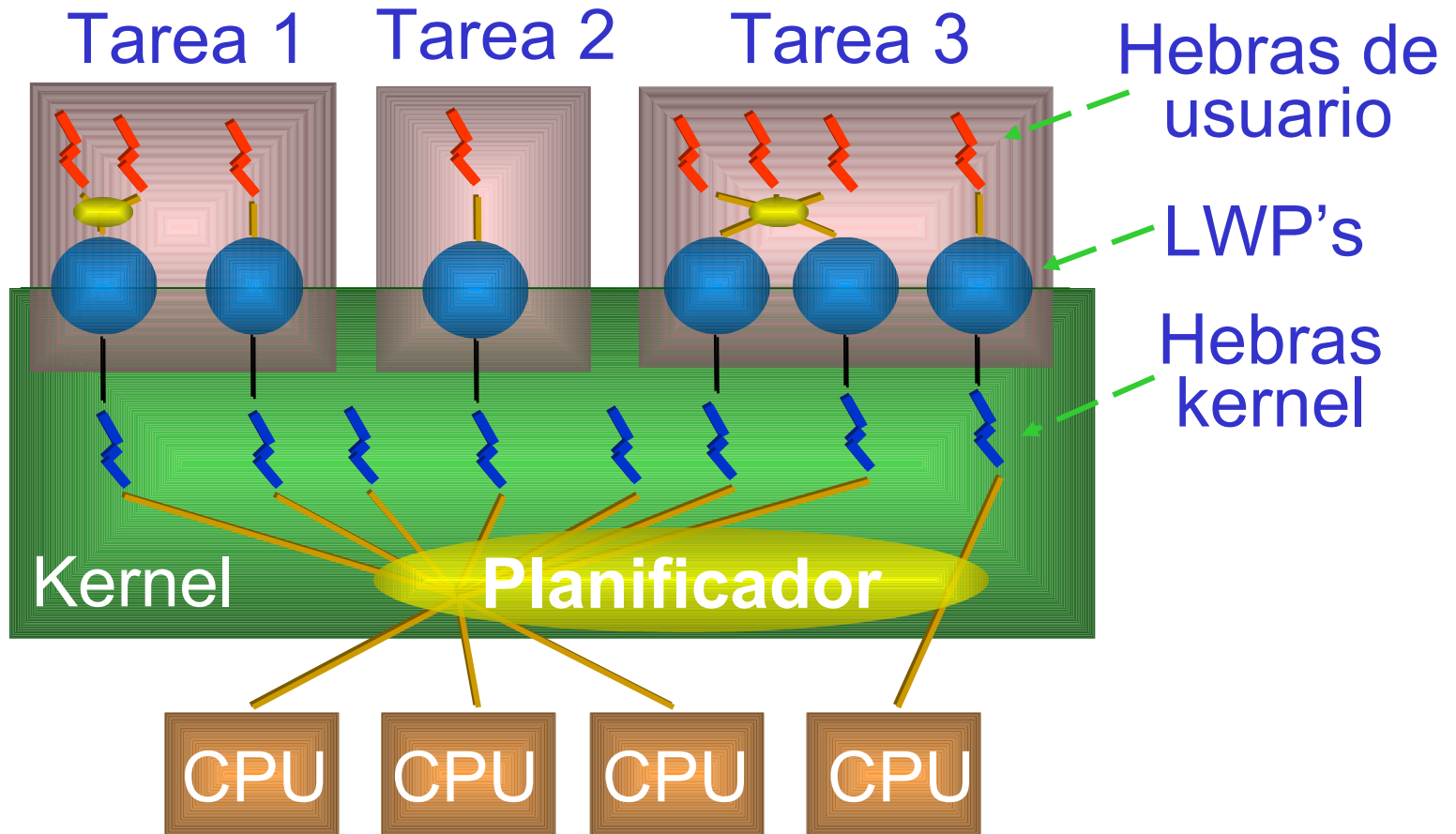


Estructura de proceso
en Solaris 2





Tareas, hebras, y LWP's





Beneficios de las hebras

- ¿Por qué no usar procesos con memoria compartida?
 - Los procesos son más caros: + coste su creación, su destrucción y cambio de contexto.
- Los procesos devoran memoria:
 - Un cientos de procesos no funcionan.
 - Es posible crear cientos de hebras.



Hebras en Linux

- Linux soporta:
 - **Hebras kernel** en labores de sistemas – ejecutan una única función en modo kernel: procesos 0 y 1, limpieza de cachés de disco, intercambio, servicio de conexiones de red, etc.
 - **Bibliotecas de hebras de usuario.**
- No todas las bibliotecas del sistema son reentrantes. *glibc v.2* lo es

<http://linas.org/linux/threads-faq.html>





Código reentrante

- Código **reentrante** es aquel que funciona correctamente si 2 ó más hebras lo ejecutan simultáneamente. Se dice también que es *thread-safe*.
- Para que sea reentrante no debe tener datos locales o estáticos.
- El SO debe ser código reentrante.
- P.ej. Linux no es 100% reentrante. MS-DOS y la BIOS no son reentrantes.



Programación con hebras



- Veamos como se realiza un programa multihebrado, utilizando como soporte las hebras de Windows.
- Vamos a dibujar varias bolas de colores moviendose en una ventana; cada bola es controlada por una hebra distinta.
- En la demostración, podemos manipular las bolas a través de las operaciones para:
 - Crear/terminar, suspender/reanudar, bloquear hebras
 - Asignar prioridades.



Ejemplo

```
// PROGRAMA: bolas.c
#include <windows.h> ...
//declaración variables compartidas
..
////////////////////////////////////
// FUNCION: DibujarBola
////////////////////////////////////

void DibujarBola (*ColorDeBola)
{
    //dibuja y mueve una bola
}
```

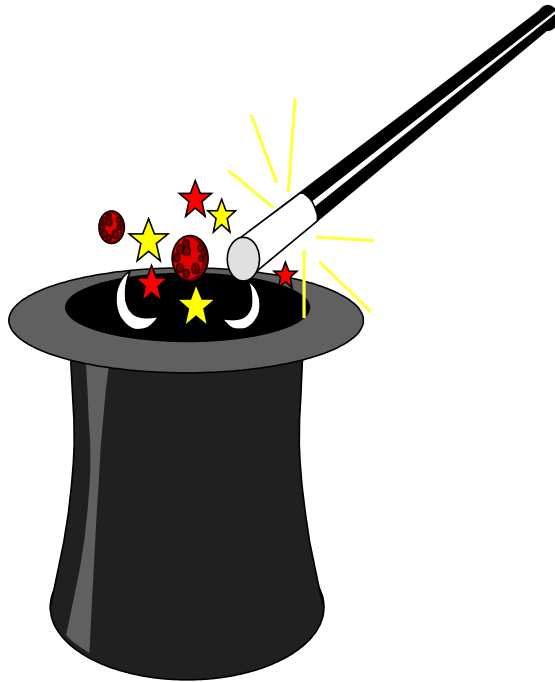



Ejemplo (y ii)

```
// FUNCION: main
////////////////////////////////////
void main()
{
HANDLE VecHeb[2];           //Descriptor de hebras
DWORD IDHebra;             //ID de la hebra
...
VecHeb[0]=CreateThread(.,DibujarBola,.,Roja,IDHebra);
VecHeb[1]=CreateThread(.,DibujarBola,.,Azul,IDHebra);
WaitForMultipleObjet(..., VectorHebras,...);
// el proceso finaliza aquí
}                          //Ejecución de programa ping-pong.
```



Operaciones sobre procesos



- Crear un proceso
- Ejecutar un programa
- Finalizar un proceso
- Sincronizar un proceso



Creación de procesos

- Los pasos a seguir por el SO:
 - Asignarle un PCB
 - Establecer su contexto de memoria (espacio de direcciones)
 - Cargar imagen (ejecutable) en memoria
 - Ajustar su contexto de CPU (registros)
 - Marcar la tarea como ejecutable:
 - a. saltar al punto de entrada, o
 - b. ponerlo en la cola de procesos preparados.



Posibilidades

- Un proceso puede crear otros procesos, y
 - Formar un árbol de procesos (UNIX) – relación de parentesco entre procesos.
 - No mantener una jerarquía (Win 2000).
- ¿Compartir recursos entre el creador y el creado?
 - Comparten todos los recursos, o un subconjunto.
 - Creador y creado no comparte recursos.



Posibilidades (cont.)

- Respecto a la ejecución:
 - Creador/creado se ejecutan concurrentemente.
 - Creador espera al que el creado termine.
- Sus espacios de direcciones son:
 - **Clonados** –se copia el espacio del creador para el creado: comparten el mismo código, datos, ej. Unix.
 - **Nuevos** –el proceso creado inicia un programa diferente al del creador. ej. Windows 2000.



Terminar un proceso

- Un proceso finaliza cuando ejecuta la declaración de terminación (explícita o implícitamente).
- Los pasos a seguir:
 - Envío de datos del proceso creado a creador. P.ej. Código de finalización.
 - El SO desasigna los recursos que tiene.



Terminación (cont.)

- El proceso puede finalizar la ejecución de otro si:
 - Ha sobrepasado los recursos asignados.
 - La tarea asignada al proceso ya no es necesaria.
 - Va a finalizar y el SO no permite que los procesos creados por él puedan continuar: terminación en cascada.



APIs de Unix y Win32

Operación	Unix	Win ³²
Crear	<code>fork()</code> <code>exec()</code>	<code>CreateProcess()</code>
Terminar	<code>_exit()</code>	<code>ExitProcess()</code>
Obtener código finalización	<code>wait</code> <code>waitpid</code>	<code>GetExitCodeProcess</code>
Obtener tiempos	<code>times</code> <code>wait^r</code> <code>wait^e</code>	<code>GetProcessTimes</code>
Identificador	<code>getpid</code>	<code>GetCurrentProcessId</code>
Terminar otro proceso	<code>kill</code>	<code>TerminateProcess</code>

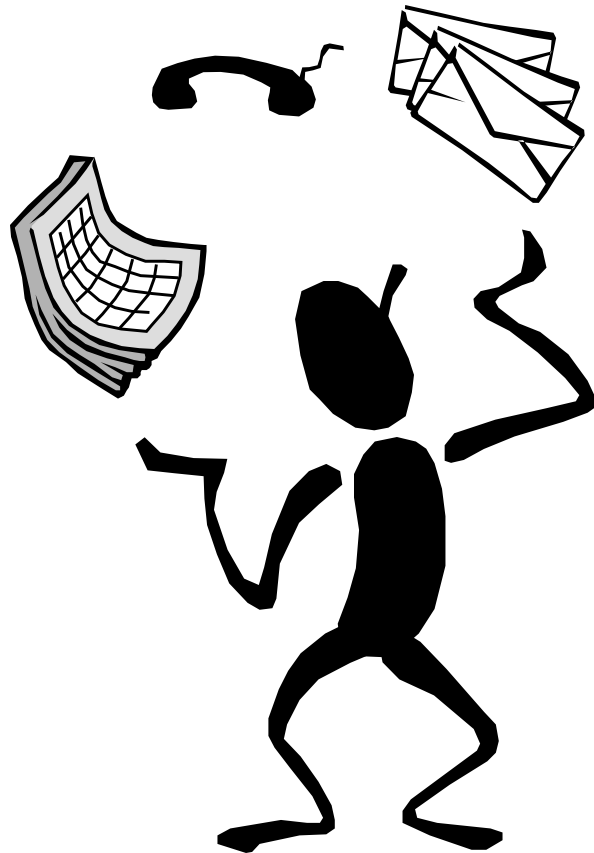


APIs de hebras

Operación	Pthread	Win ³²
Crear	<code>Pthread_create</code>	<code>CreateThread</code>
Crear en otro proceso	-	<code>CreateRemoteThread</code>
Terminar	<code>Pthread_exit</code>	<code>ExitThread</code>
Código finalización	<code>Pthread_yield</code>	<code>GetExitCodeThread</code>
Terminar	<code>Pthread_cancel</code>	<code>TerminateThread</code>
Identificador	-	<code>GetCurrentThreadId</code>



Planificación



- Planificador: definición y tipos.
- Algoritmos de planificación:
 - FIFO
 - SJF
 - Prioridades
 - Round-robin
 - Colas múltiples
- Planificación en tiempo-real.



Planificadores de procesos

- **Planificador** – módulo del SO que controla la utilización de un recurso.
- **Planificador a largo plazo** (o *de trabajos*) – selecciona procesos que deben llevarse a la cola de preparados; transición (1) en transparencia 20
- **Planificador a corto plazo** (o *de la CPU*) – selecciona al proceso que debe ejecutarse a continuación, y le asigna la CPU; transición (2).



Características de los planificadores

- **Planificador a corto plazo:** se invoca con mucha frecuencia (del orden de mseg.) por lo que debe ser rápido en su ejecución.
- **Planificador a largo plazo:** se invoca con menor frecuencia (segundos o minutos) por lo que puede ser lento.
- Controla el **grado de multiprogramación** (n° de trabajos en memoria principal).





Planificador a medio plazo

- En algunos SO's, p. ej. tiempo compartido, es a veces necesario sacar procesos de memoria (reducir el grado de multiprogramación) por cambios en los requisitos de memoria, y luego volverlos a introducir (*intercambio* -*swapping*). El **planificador a medio plazo** se encarga de devolver los procesos a memoria; transición (3).
- Es un mecanismo de gestión de memoria que actúa como planificador.



Ráfagas de CPU

- La ejecución de un proceso consta de ciclos sucesivos **ráfagas de CPU-E/S**.
- Procesos acotados por E/S – muchas ráfagas cortas de CPU.
- Procesos acotados por CPU – pocas ráfagas largas de CPU.
- Procesos de Tiempo-real – ejecución definida por (repetidos) plazos (*deadline*). El procesamiento por plazo debe ser conocido y acotado.



¿Cuándo planificar?

Políticas de planificación

- Las decisiones de planificación pueden tener lugar cuando se conmuta del :
 - Estado ejecutándose a bloqueando.
 - Estado ejecutándose a preparado.
 - Estado bloqueado a preparado.
 - Estado ejecutándose a finalizado.
- Es decir, siempre que un proceso abandona la CPU, o se inserta un proceso en la cola de preparados.



Políticas de planificación

- **Planificación apropiativa** (*preemptive*): El SO puede quitar la CPU al proceso. Casos 2 y 3.
- **Planificación no apropiativa** (*no preemptive*): No se puede retirar al proceso de la CPU, este la libera voluntariamente al bloquearse o finalizar. Casos 1 y 4.
- La elección entre ambas depende del comportamiento de la aplicación (p.e. RPC rápida) y del diseño que queramos hacer del sistema.



Apropiación frente a No-apropiación

- La apropiación nos asegura que un trabajo no bloquea a otro igualmente importante.
- Cuestiones a tener en cuenta:
 - ¿Cuándo apropiar? ¿en tiempo de interrupción?
 - ¿Tamaño de la fracción de tiempo? Afecta al tiempo de respuesta y a la productividad.



Apropiación frente a No apropiación

- La planificación no apropiativa requiere que los trabajos invoquen explícitamente al planificador.
- Un trabajo erróneo puede tirar el sistema.
- Simplifica la sincronización de hebras/procesos.



Despachador

- El **despachador** (*dispatcher*) da el control de la CPU al proceso seleccionado por el planificador. Realiza lo siguiente:
 - Cambio de contexto (en modo kernel).
 - Conmutación a modo usuario.
 - Salto a la instrucción del programada para su reanudación.
- **Latencia de despacho** – tiempo que emplea el despachador en detener un proceso y comenzar a ejecutar otro.



El bucle de despacho

- En pseudocódigo:

```
while (1) { /* bucle de despacho */  
    ejecutar un proceso un rato;  
    parar al proceso y salva su estado;  
    carga otro proceso;  
}
```

- ¿Cómo obtiene el despachador el control ?
 - **Síncrona** – un proceso cede la CPU.
 - **Asíncrona** – iniciado por una interrupción u ocurrencia de un evento que afecta a un proceso (p. ej. fin de e/s, liberación recurso,...)



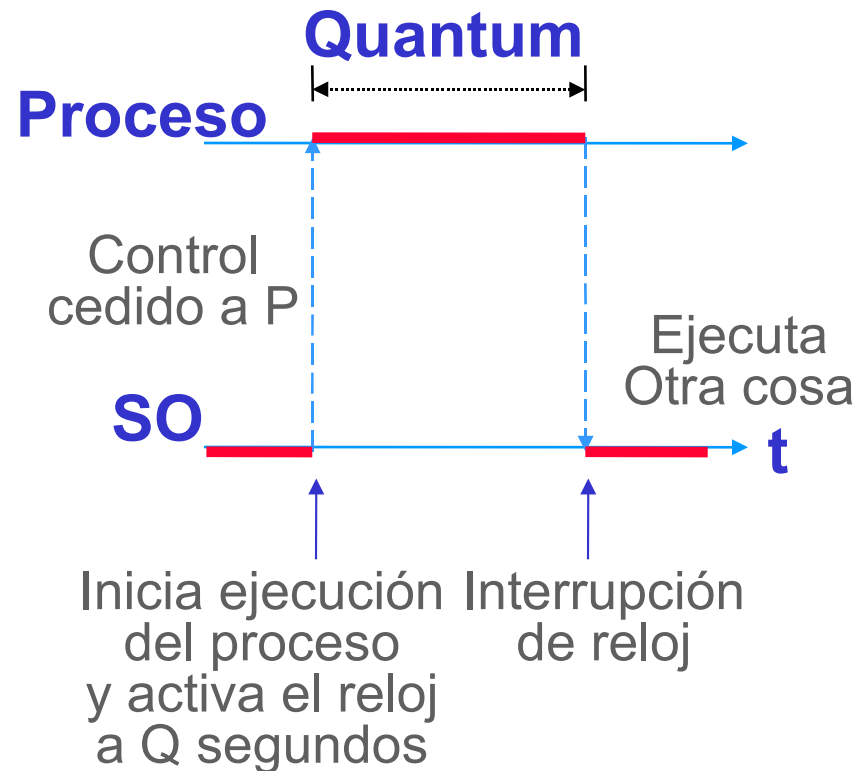
Gestor de interrupciones: revisitado

- Realiza las siguientes operaciones:
 - Salva el contexto del proceso en ejecución.
 - Determina el tipo de interrupción y ejecuta la rutina de servicio de interrupción adecuada.
 - **Selecciona el proceso que se ejecuta a continuación.**
 - **Restaura el contexto salvado del seleccionado para ejecutarse**



Implementación del tiempo-compartido

- El SO asigna la CPU a un proceso y le asocia una fracción de tiempo (*time-slice*) o *quantum*.
 - En cada tick de reloj, la ISR de reloj comprueba si el plazo a concluido: Si, el control se devuelve al SO; no, sigue el proceso.





Mecanismos frente a políticas

- Un mecanismo es el código (a menudo de bajo nivel) que manipula un recurso.
 - CPU: cambio entre procesos ...
 - Memoria: asignar, liberar,..
 - Disco: leer, escribir, ..
- Una política decide “cómo, quién, cuando y porqué”
 - Cuanta CPU obtiene un proceso
 - Cuanta memoria le damos
 - Cuando escribir en disco



Un tema recurrente

- Los diseñadores de SOs tratan de mantener separados los mecanismos de las políticas:
 - Facilita la reutilización de mecanismos (p. ej., controladores de dispositivos)
 - Permite la adaptación de políticas.
- Un tema común de investigación es la separación de políticas/mecanismos
 - En algunos casos, el mecanismo de uno es la política de otro.
 - Realmente reduce el nivel de abstracción.



Criterios de planificación

- **Utilización** –mantener la CPU tan ocupada como sea posible.
- **Productividad** –nº de procesos que completan su ejecución por unidad de tiempo.
- **Tiempo de retorno** –cantidad de tiempo necesaria para ejecutar un proceso dado.
- **Tiempo de espera** –tiempo que un proceso ha estado esperando en la cola de preparados.
- **Tiempo de respuesta** –tiempo que va desde que se remite una solicitud hasta que se produce la primera respuesta (no salida).



Métricas de planificación

- Máxima utilización.
- Máxima producción.
- Mínimo tiempo de retorno.
- Mínimo tiempo de espera.
- Mínimo tiempo de respuesta.
 - La elección depende del tipo de aplicaciones y el uso del SO.



En la mayoría de los casos, algunos de estos criterios son contrapuestos.



Primero en Llegar, Primero en ser Servido (FIFO)

■ Para los procesos de la tabla, construimos el Diagrama Gantt.

■ Tiempos de espera:

■ $P_1 = 0$

■ $P_2 = 24$

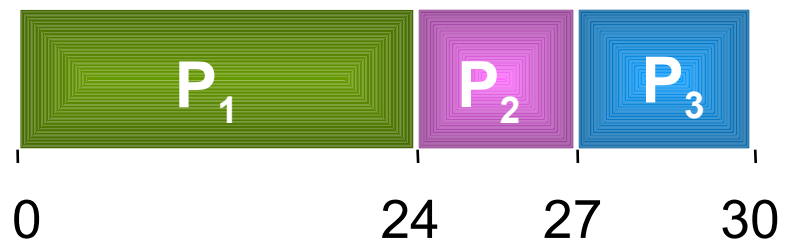
■ $P_3 = 27$

■ Tiempo medio de espera (t_e medio):

$$(0 + 24 + 27) / 3 = 17$$

Proceso	T. ráfaga	T. llegada
P_1	24	0
P_2	3	0
P_3	3	0

b)



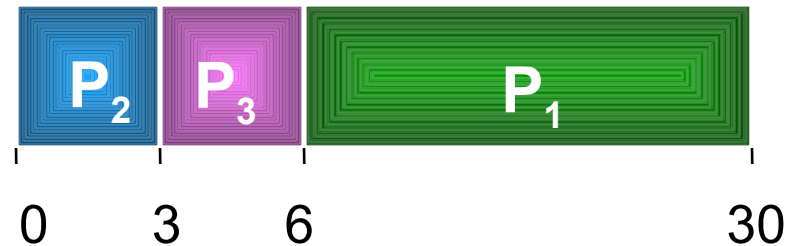


Efecto escolta

- Si cambiamos el orden de ejecución de los procesos, t_e medio es mejor:

$$(6+0+3)/3 = 3$$

- **Efecto escolta** – los procesos cortos esperan a los largos.



Tiempos de espera:

$$P_1 = 6$$

$$P_2 = 0$$

$$P_3 = 3$$



Primero el Más Corto (SJF)

- Se planifica al proceso cuya siguiente ráfaga es la más corta. Tiene dos versiones:
 - No apropiativa –el proceso en ejecución no se apropia hasta que complete su ráfaga.
 - Apropiativa o **Primero el de tiempo restante menor (SRTF)** –un proceso con ráfaga más corta que el tiempo restante del proceso en ejecución, apropia al proceso actual.



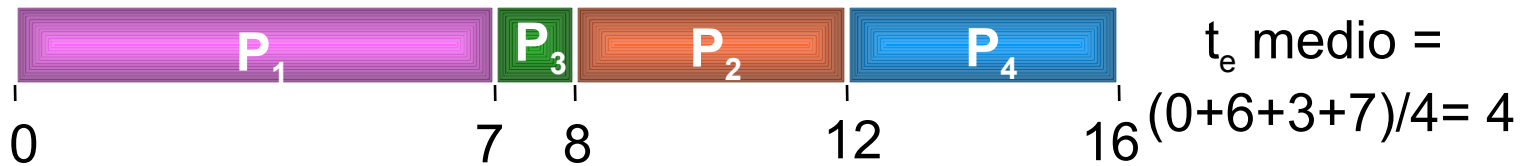
Características de SJF

- Minimiza el t_e medio para un conjunto dado de procesos (no así, el tiempo de respuesta).
- Se comporta como un FiFo, si todos los procesos tienen la misma duración de ráfaga.
- Actualmente se utilizan variantes de este algoritmo para planificación de procesos en tiempo-real.

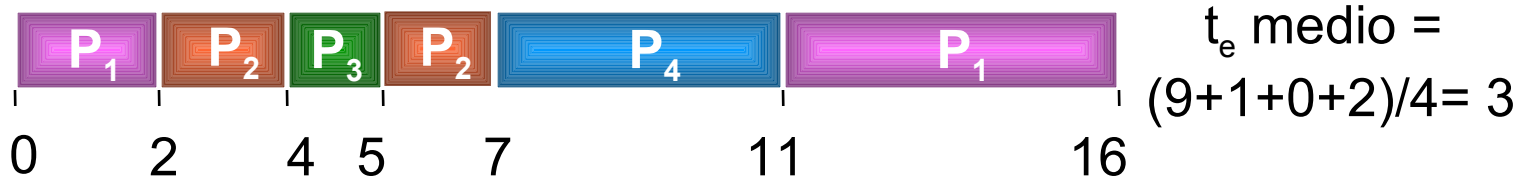
Ejemplo de SJF

Proceso	T. Ráfaga	T. Llegada
P ₁	7	0
P ₂	4	2
P ₃	1	4
P ₄	4	5

❑ SJF (no apropiativo)



❑ SRTF (apropiativo)






¿Cuánto dura la siguiente ráfaga?

- Estimamos su duración con ráfagas previas.
- Sean:
 - T_n =duración actual de la n-ésima ráfaga
 - Ψ_n =valor estimado de la n-ésima ráfaga
 - Un peso W , donde $0 \leq W \leq 1$
 - Definimos: $\Psi_{n+1} = W * T_n + (1-W) \Psi_n$
- $W=0$; $\Psi_{n+1} = \Psi_n$, no influye historia reciente.
- $W = 1$; $\Psi_{n+1} = T_n$, sólo cuenta la ráfaga actual.



Planificación por prioridades

 No todos los procesos son iguales, asociamos a cada proceso un número entero que indique su “importancia”, y damos la CPU al proceso con mayor prioridad.

- Puede ser:– apropiativo o no apropiativo.
 - estático o dinámico.



Inanición –los procesos de baja prioridad pueden no ejecutarse nunca.

- **Envejecimiento** –incremento de prioridad con el paso del tiempo.



Planificación Round-robin (RR)

- A cada proceso se le asigna un **quantum** de CPU (valores típicos 10–100 ms). Pasado este tiempo, si no ha finalizado ni se ha bloqueado, el SO apropia al proceso y lo pone al final de la cola de preparados.
- Recordar transparencia [42 del Tema 1](#), donde se mostraba con se controla el tiempo del quantum mediante el reloj.



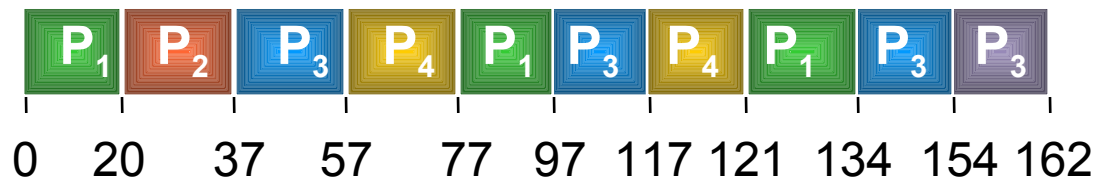
Características de RR

- Realiza una asignación imparcial de la CPU entre los diferentes trabajos.
- Tiempo de espera medio:
 - bajo si la duración de los trabajos varía.
 - Malo si la duración de los trabajos es idéntica.
- El rendimiento depende del tamaño de q :
 - q grande \Rightarrow FIFO.
 - q pequeño \Rightarrow mucha sobrecarga si q no es grande respecto a la duración del cambio de contexto.

Ejemplo de RR con $q = 20$

Proceso	T. Ráfaga	T. Llegada
P ₁	53	0
P ₂	17	0
P ₃	68	0
P ₄	24	0

El diagrama de Gantt es



Típicamente, tiene un mayor tiempo de retorno que SRT, pero mejor *respuesta*.



Colas múltiples

- La cola de preparados se fracciona en varias colas; cada cola puede tener su propio algoritmo de planificación.
- Ahora, hay que realizar una **planificación entre colas**. Por ejemplo:
 - Prioridades fijas entre colas.
 - Tiempo compartido – cada cola obtiene tiempo de CPU que reparte entre sus procesos.



Ejemplos

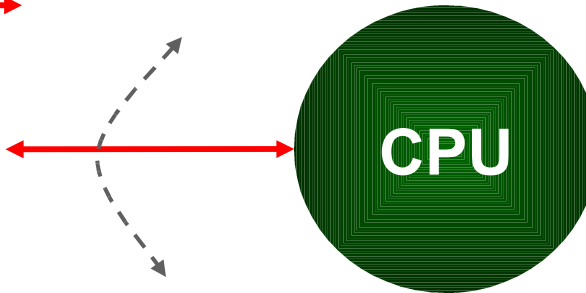
Procesos interactivos



1. Por prioridades: procesos interactivos primero.



Procesos batch



1. Tiempo compartido: 80% CPU Cola1, y 20% para Cola2.



Colas múltiples con realimentación

- Como el anterior, pero ahora un proceso puede moverse entre varias colas. Así, puede implementarse envejecimiento.
- Parámetros que definen el planificador:
 - Número de colas.
 - Algoritmo de planificación para cada cola.
 - Métodos utilizados para determinar:
 - ◆ Cuando ascender de cola a un proceso.
 - ◆ Cuando degradar de cola a un proceso.
 - ◆ Cola de entrada de un proceso que necesita servicio.



Ejemplo:

- Suponemos un sistema con tres colas:
 - Q_0 - RR con $q=8$ milisegundos.
 - Q_1 - RR con $q=16$ milisegundos.
 - Q_2 - FIFO.
- Planificación:
 - Un trabajo entra en Q_0 ; sino acaba en 8ms se mueve a Q_1 .
 - En Q_1 recibe 16 ms adicionales. Si aún no ha finalizado, es apropiado y
 - llevado a Q_2 , hasta que finaliza.



CMR en sistemas interactivos

- Planificación entre colas por prioridades.
- Asignan prioridad a los procesos basándose en el uso que hacen de CPU-E/S que indica cómo estos “cooperan” en la compartición de la CPU:
 - +prioridad cuando se realizan más E/S.
 - -prioridad cuanto más uso de CPU.
- Implementación: si un trabajo consume su cuanto, prioridad--; si se bloquea, prioridad++.



Planificación en multiprocesadores

- Con varias CPUs, la planificación es más compleja: puede interesar que una hebra se ejecute en un procesador concreto (puede tener datos en caché) o que varias hebras de una tarea planificadas simultáneamente.
- Dos posibles técnicas para repartir trabajo:
 - Distribución de carga –repartir la carga entre CPUs para no tener ninguna ociosa.
 - Equilibrio de carga –reparto uniforme de la carga entre las diferentes CPUs.



Planificación en Tiempo-Real

- *Sistemas de tiempo-real duros (hard)* – necesitan completar una tarea crítica dentro de un intervalo de tiempo garantizado. Necesitan reserva de recursos.
- *Sistemas de tiempo-real blandos (soft)* – requieren que los procesos críticos reciban prioridad sobre los menos críticos. Basta con que los procesos de tiempo-real tengan mayor prioridad y que esta no se degrade con el tiempo.



Planificación en Tiempo-Real

- Es importante reducir la latencia de despacho para acotar el tiempo de respuesta:
- Como algoritmos de planificación se suelen utilizar variaciones del SJF:
 - **EDF** (*Earliest-deadline First*) – Divide los trabajos por plazos, selecciona primero el trabajo con el plazo más próximo
 - **Razón monótona** – asigna prioridades inversamente al periodo.



Latencia de despacho



- ❑ Apropiar al proceso en ejecución, y liberar los recursos usados por el proceso de baja prioridad, y necesitados por el de alta prioridad.
- ❑ Si no se liberan se puede producir **inversión de prioridad**.