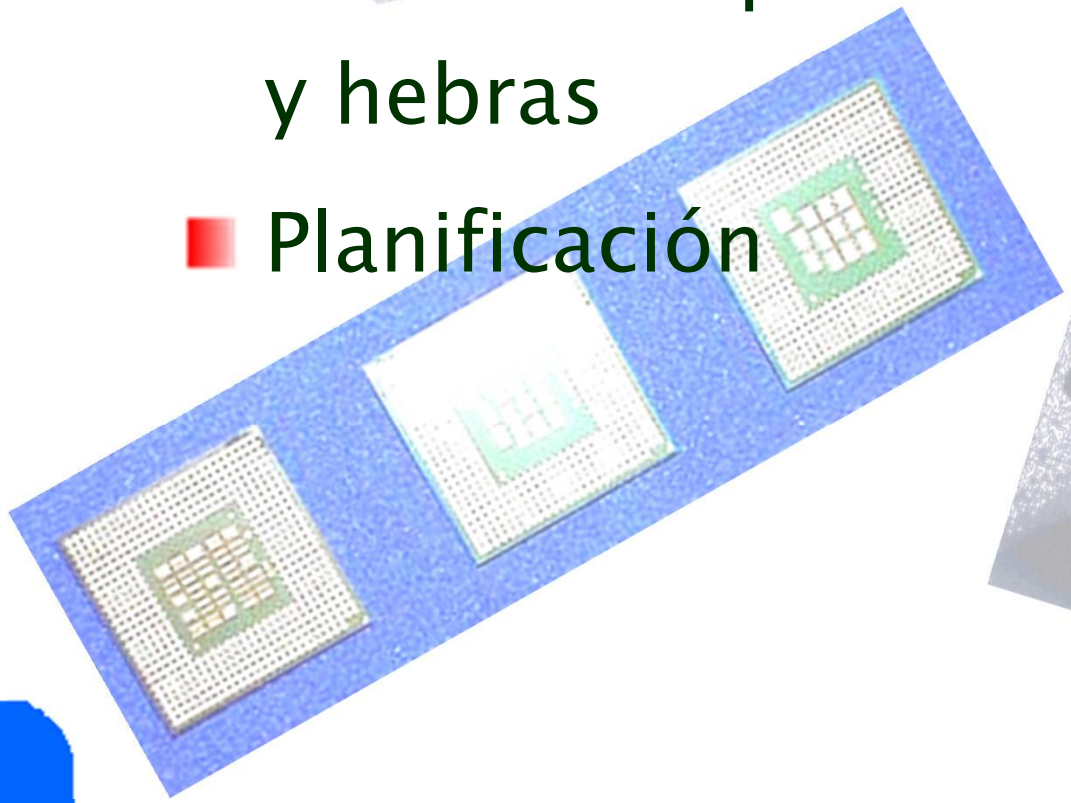
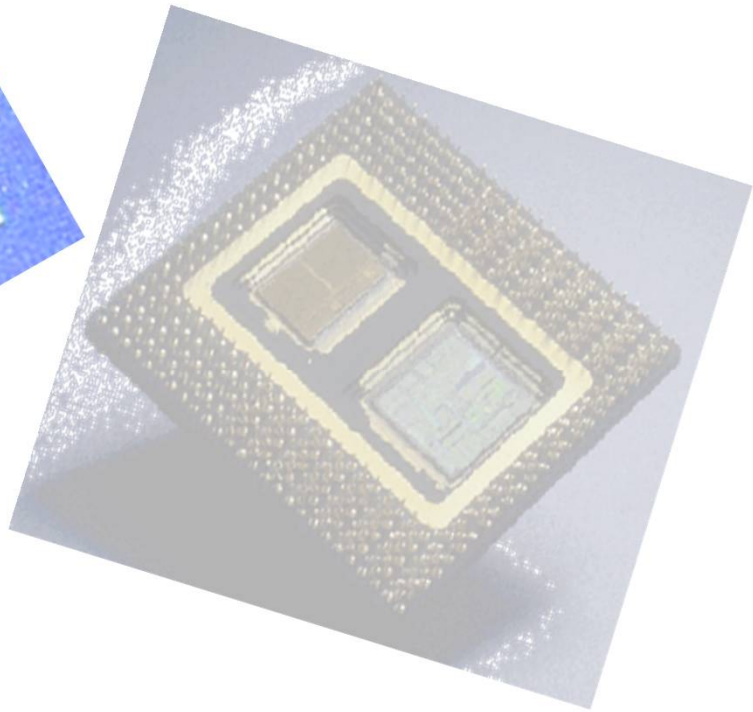
2 Procesos y hebras

- Diseño e implementación de procesos y hebras
- Planificación





Diseño

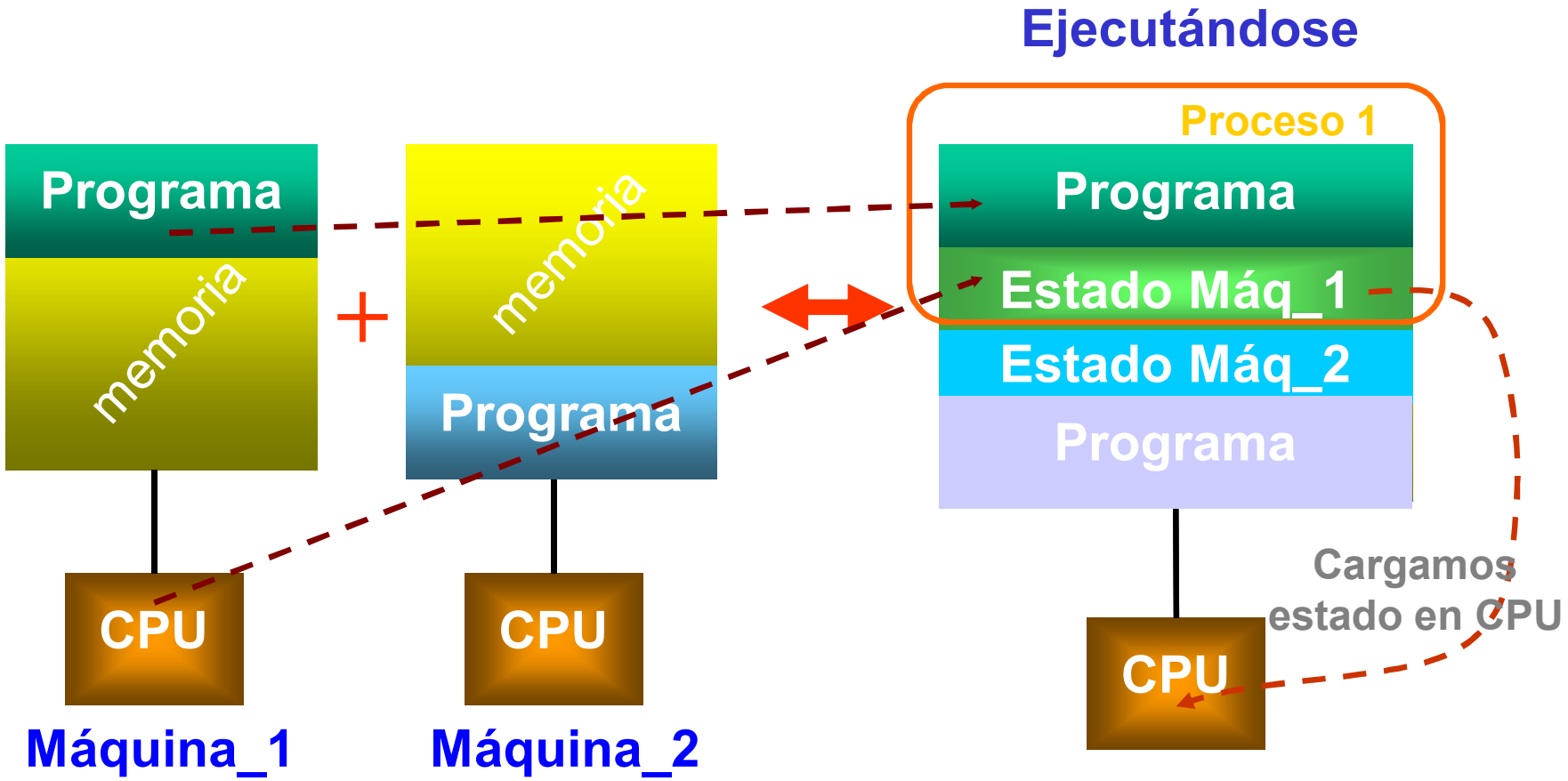
- Podríamos construir el SO como:

```
for (;;)   {
     if (aplicación())          Ejecutar();
     if (MensajeRed())          ObtenMensaje();
     if (TeclaPulsada)()        ObtenTecla();
     if (BlokeDiscoListo())     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).

Idea

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

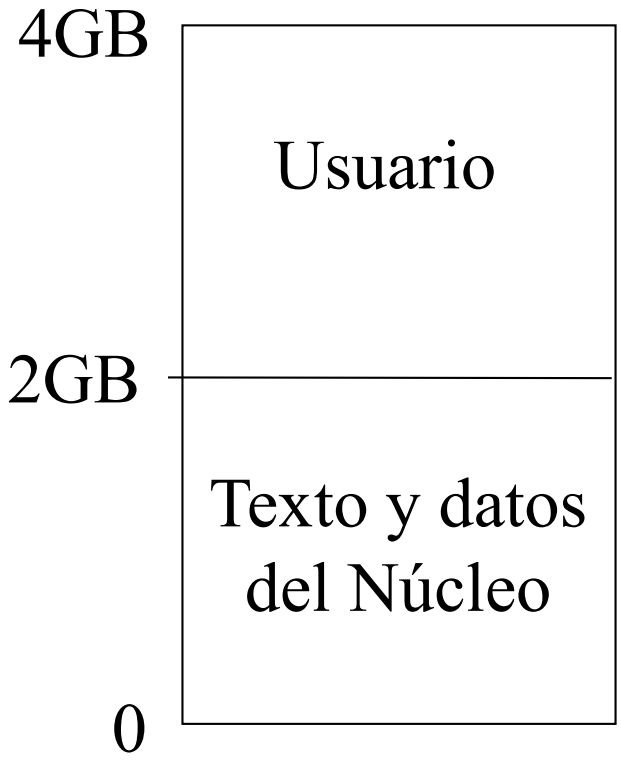


Partes de un proceso

- **Partes de un programa ejecutable:**
  - conjunto de cabeceras
  - texto del programa
  - datos inicializados y no inicializados (*bss*)
  - otras secciones (ej. tabla de símbolos)
- El núcleo divide el proceso en **regiones** (segmentos):
  - región de texto: sólo lectura
  - región de datos
  - región de pila

Diseño de Memoria

- La estructura del espacio de memoria virtual de un proceso se divide en dos partes:

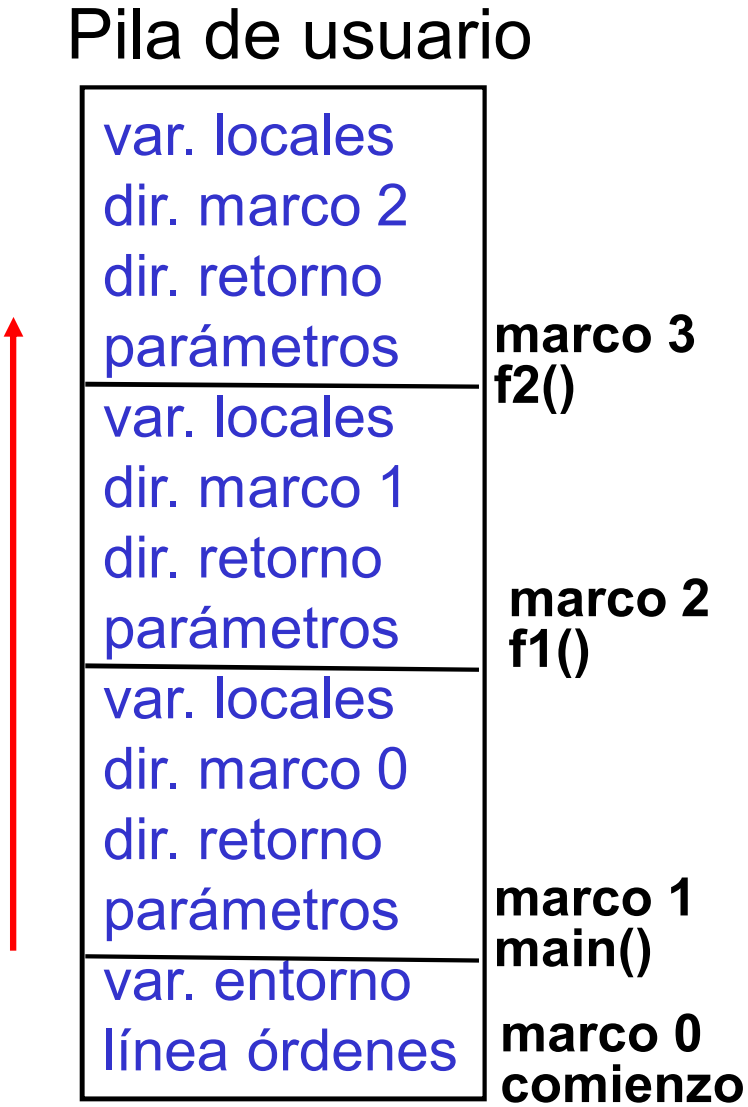


– Con direcciones de 32 bits, el espacio virtual máximo para un proceso es de 4GB

– Los primeros 2GB se ocupan con el código y las estructuras de datos del núcleo

Partes de un proceso

- La pila de usuario se crea automáticamente y su tamaño se ajusta dinámicamente
- Consta de **marcos de pila** lógicos que se insertan cuando se llama a una función y se eliminan cuando finaliza
- Como un proceso puede ejecutarse en modo supervisor y modo usuario, existe **una pila para cada modo**



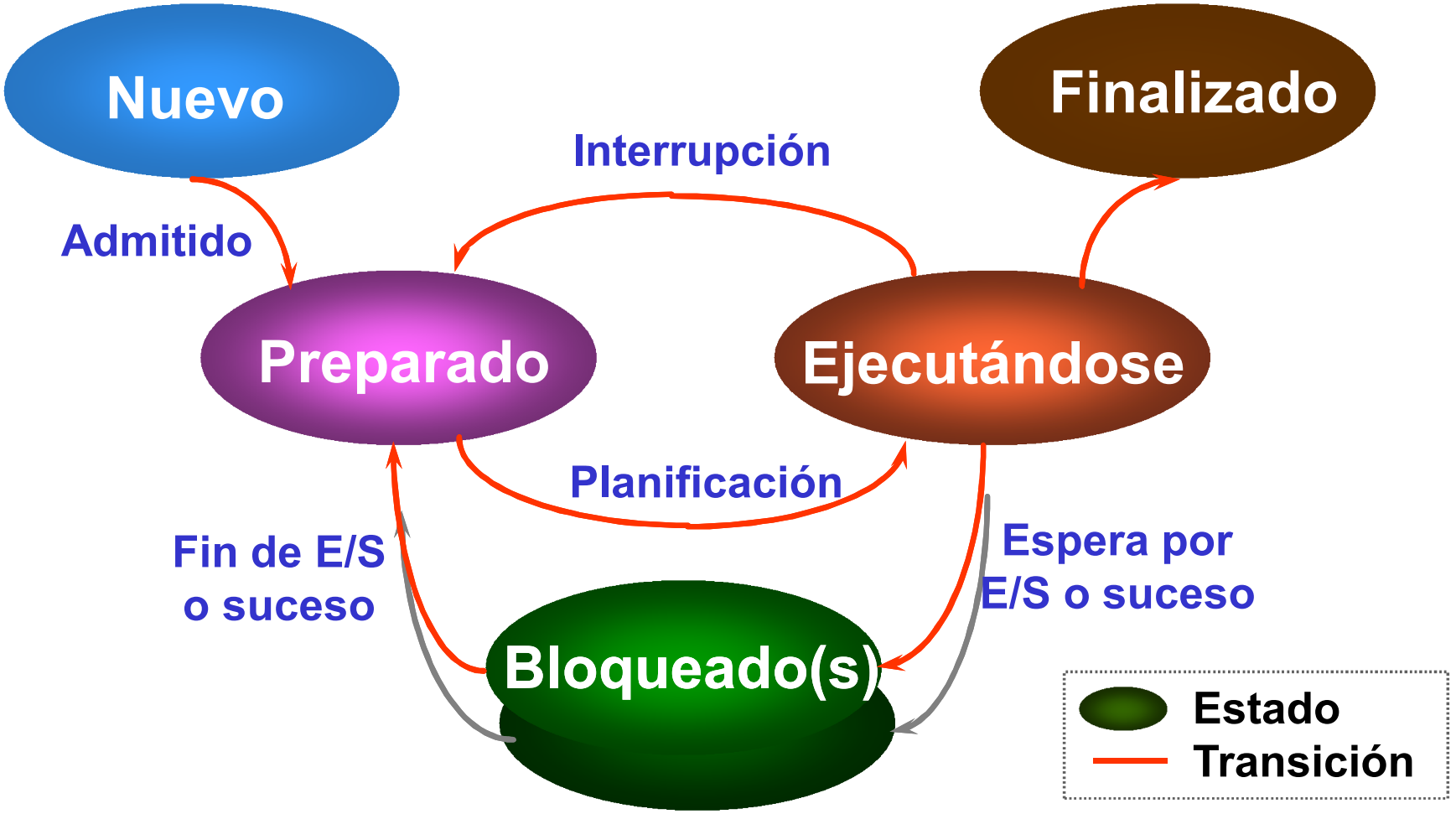
¿Qué hay en un proceso?

- 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 y su auditoría (memoria, archivos abiertos, E/S, etc.).

Estado de un proceso

- Varios procesos pueden ejecutar incluso el mismo programa (p.ej. un editor) pero cada 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



Ej.: Proceso en primer plano

```
% gcc programa.c
% _
```

Ej.: Proceso de fondo

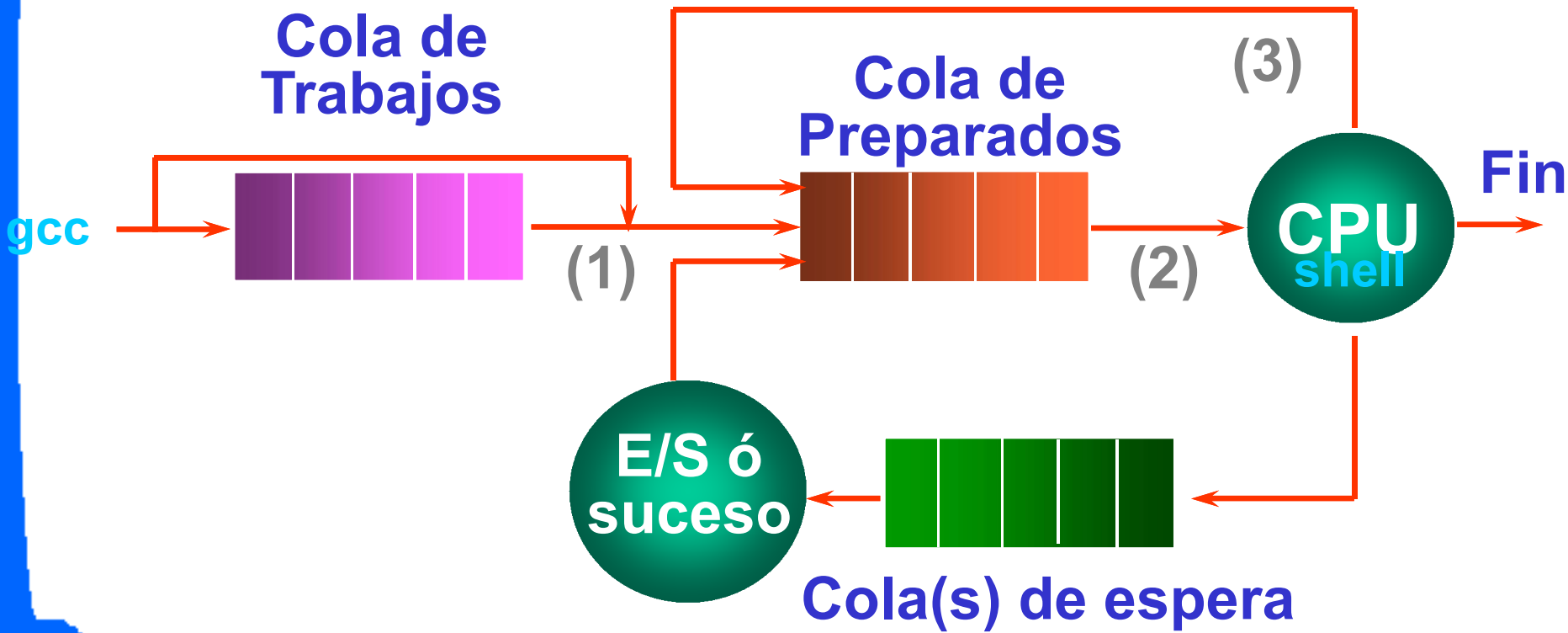
```
% gcc programa.c &
% cat file.txt
```

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:

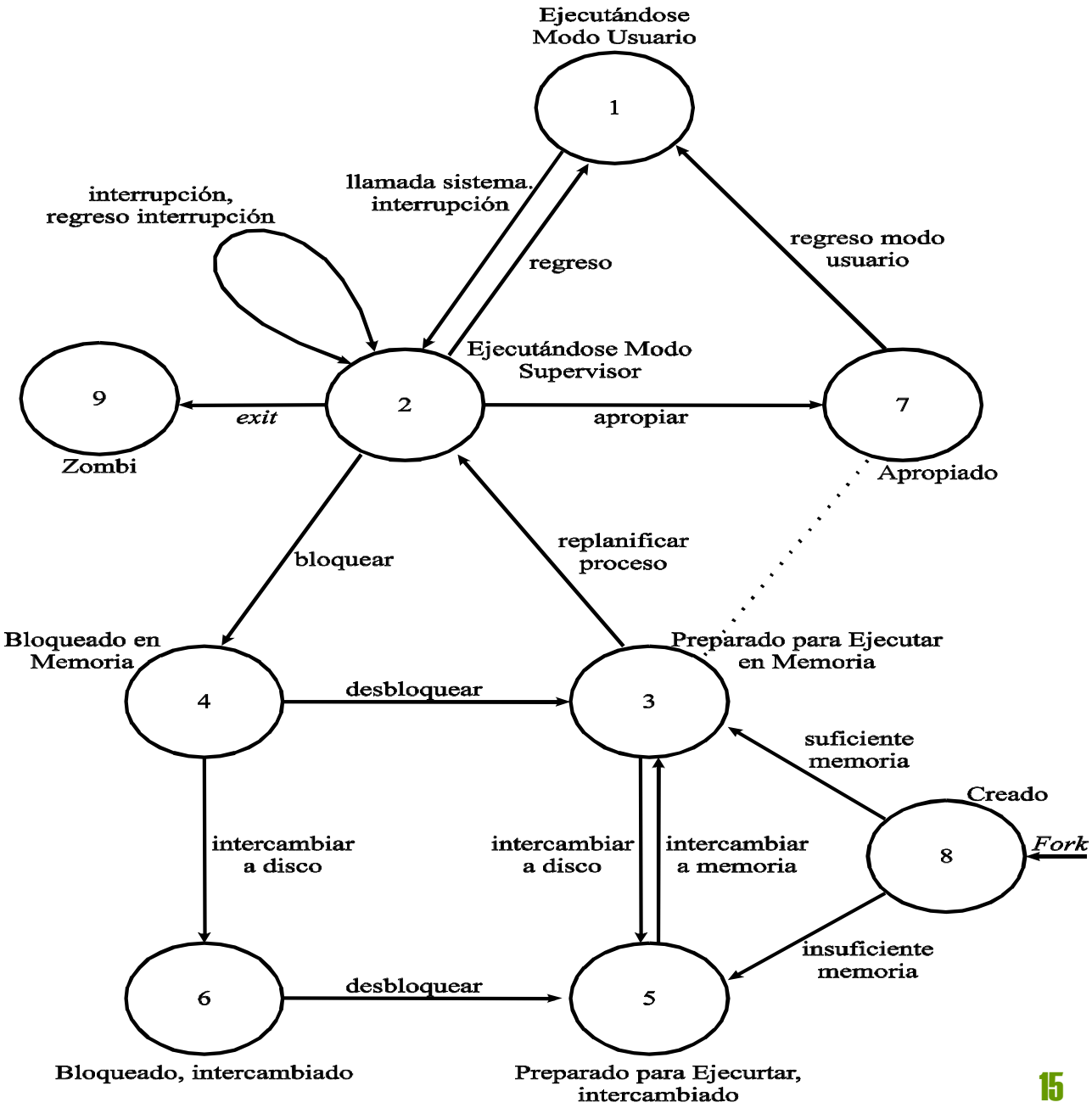


Colas de estados

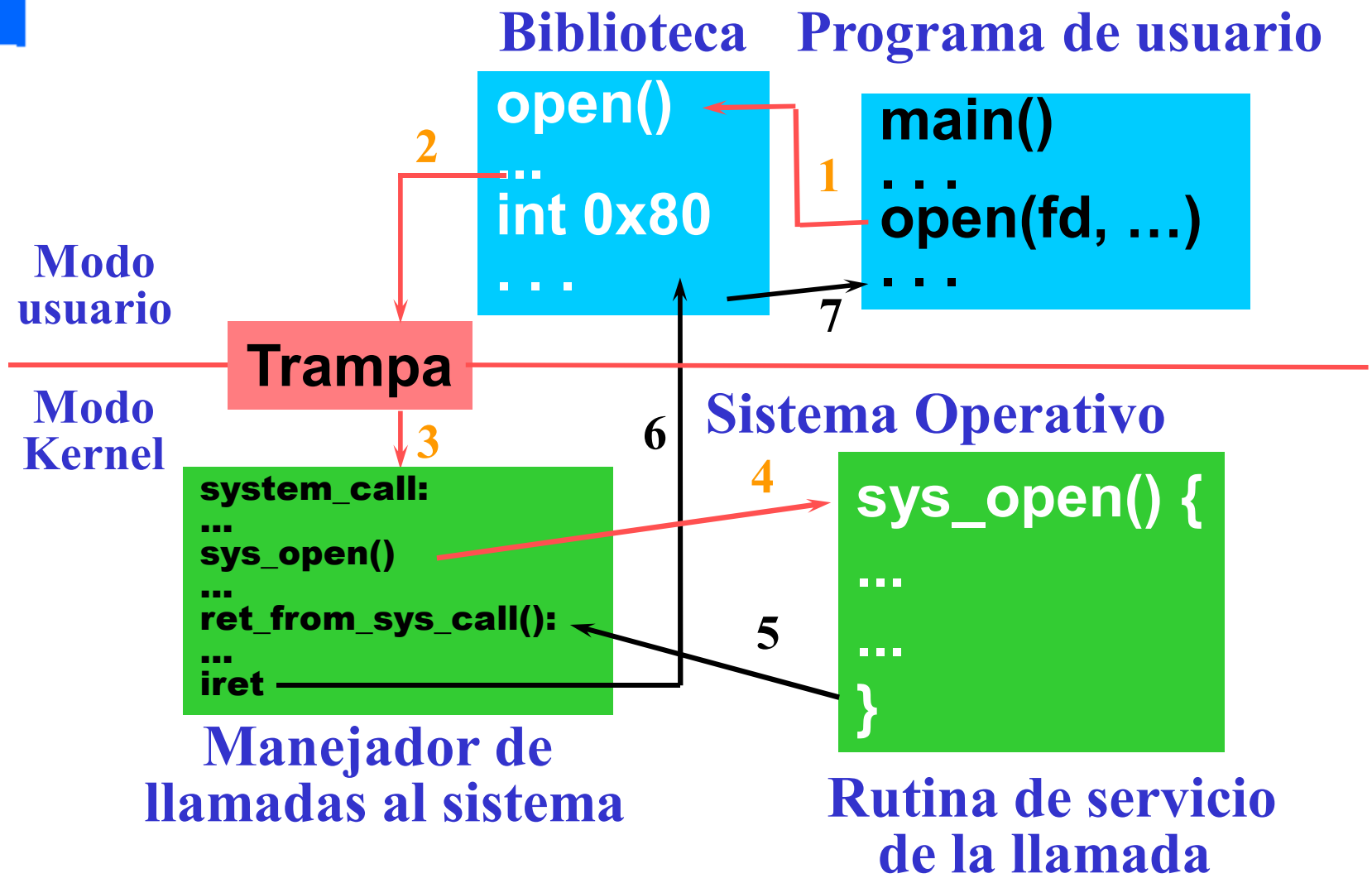
- Cola de trabajos – conjunto de todos los procesos del sistema.
- Cola de preparados – conjunto de todos los procesos que residen en memoria principal, preparados y esperando para ejecutarse.
- Cola(s) de espera – conjunto de todos los procesos esperando por un dispositivo de E/S particular o por un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

Estados/transiciones en Unix

- El proceso tiene control sobre algunas transiciones
- Ningún proceso puede desplazar a otro que se está ejecutando en modo supervisor
- El 3 y el 7 son realmente el mismo estado



Llamadas al sistema



Cambio de contexto

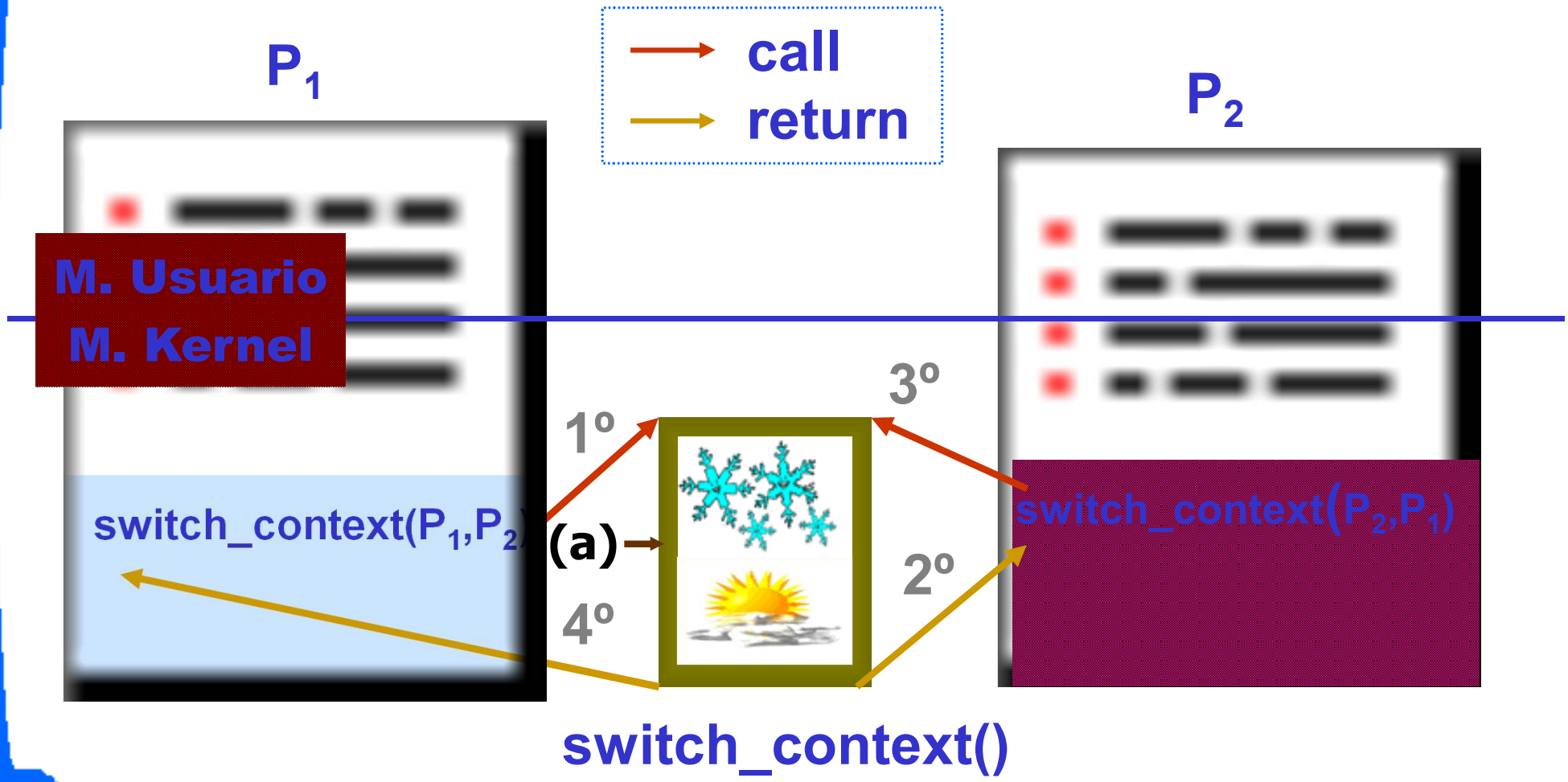
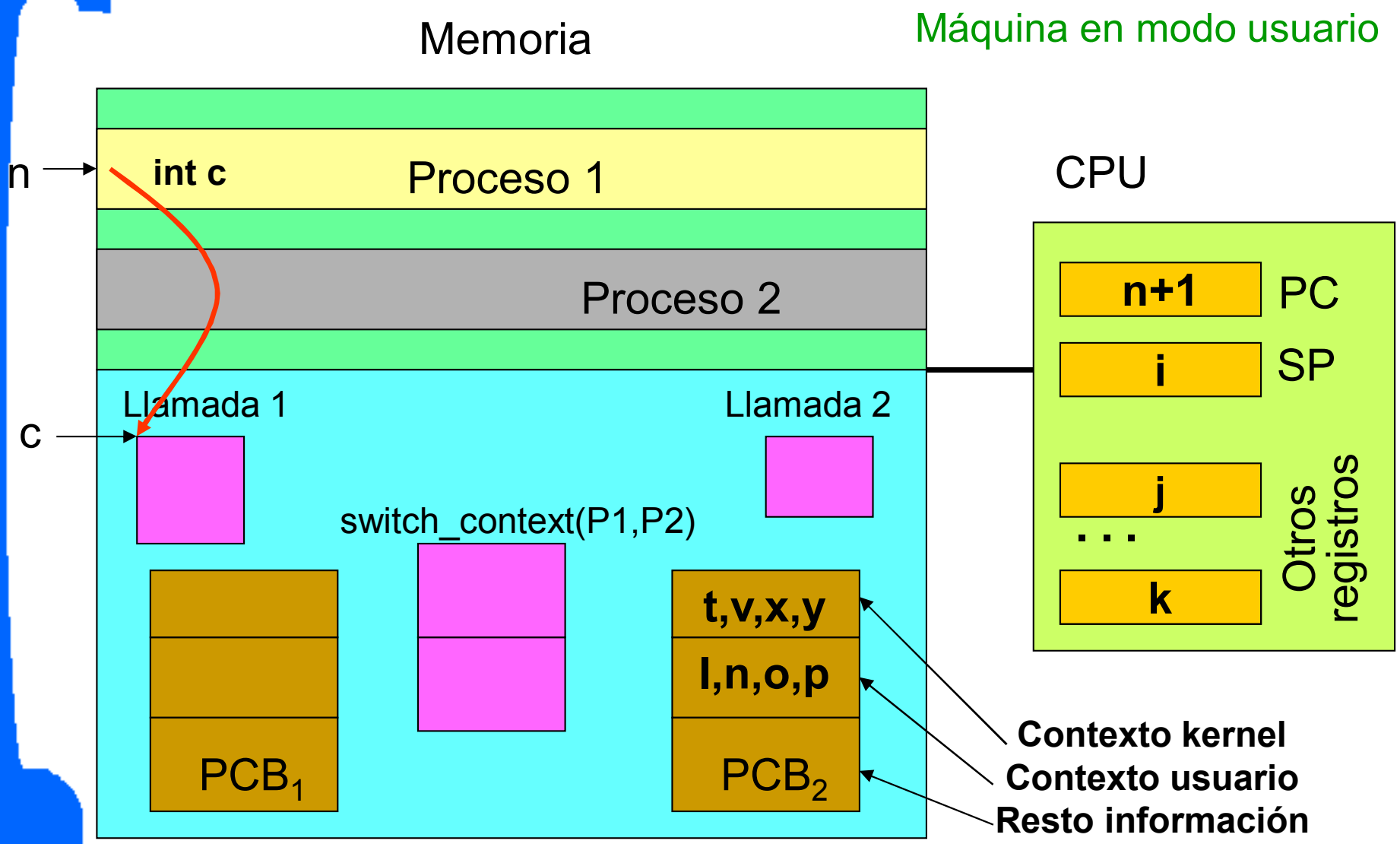


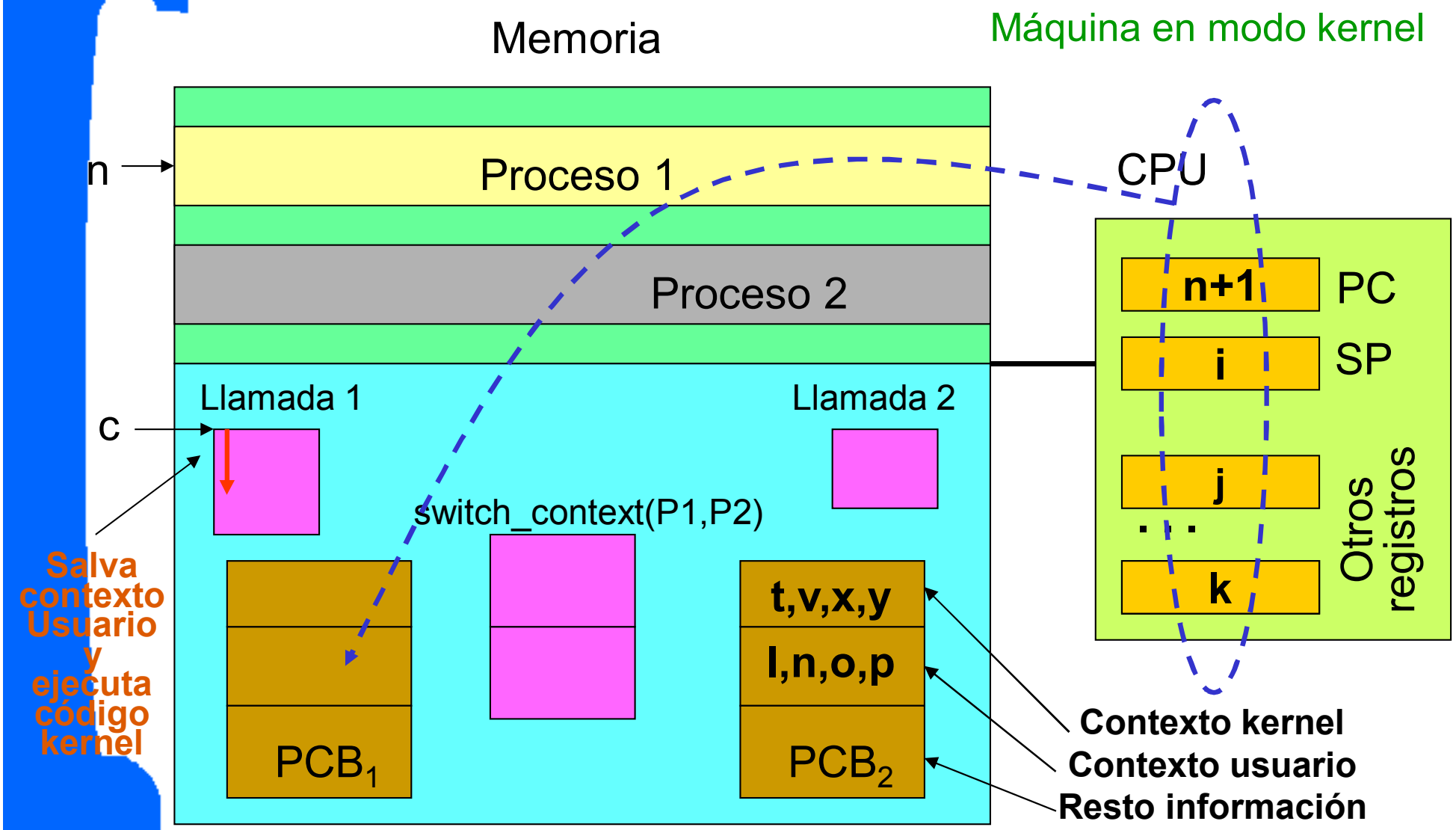
Ilustración del cambio de contexto: supuestos

- 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

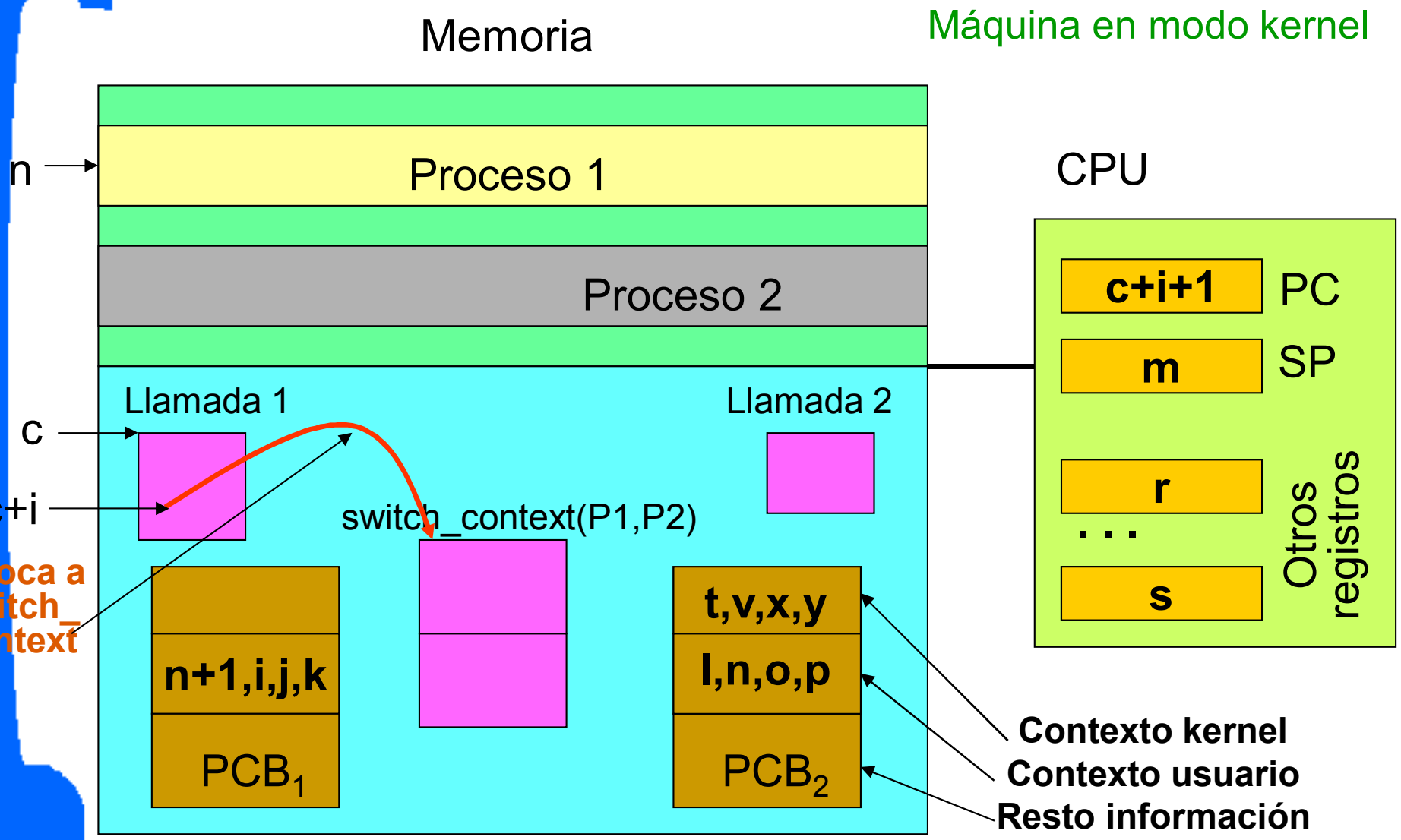
1º - P_1 ejecuta *n*



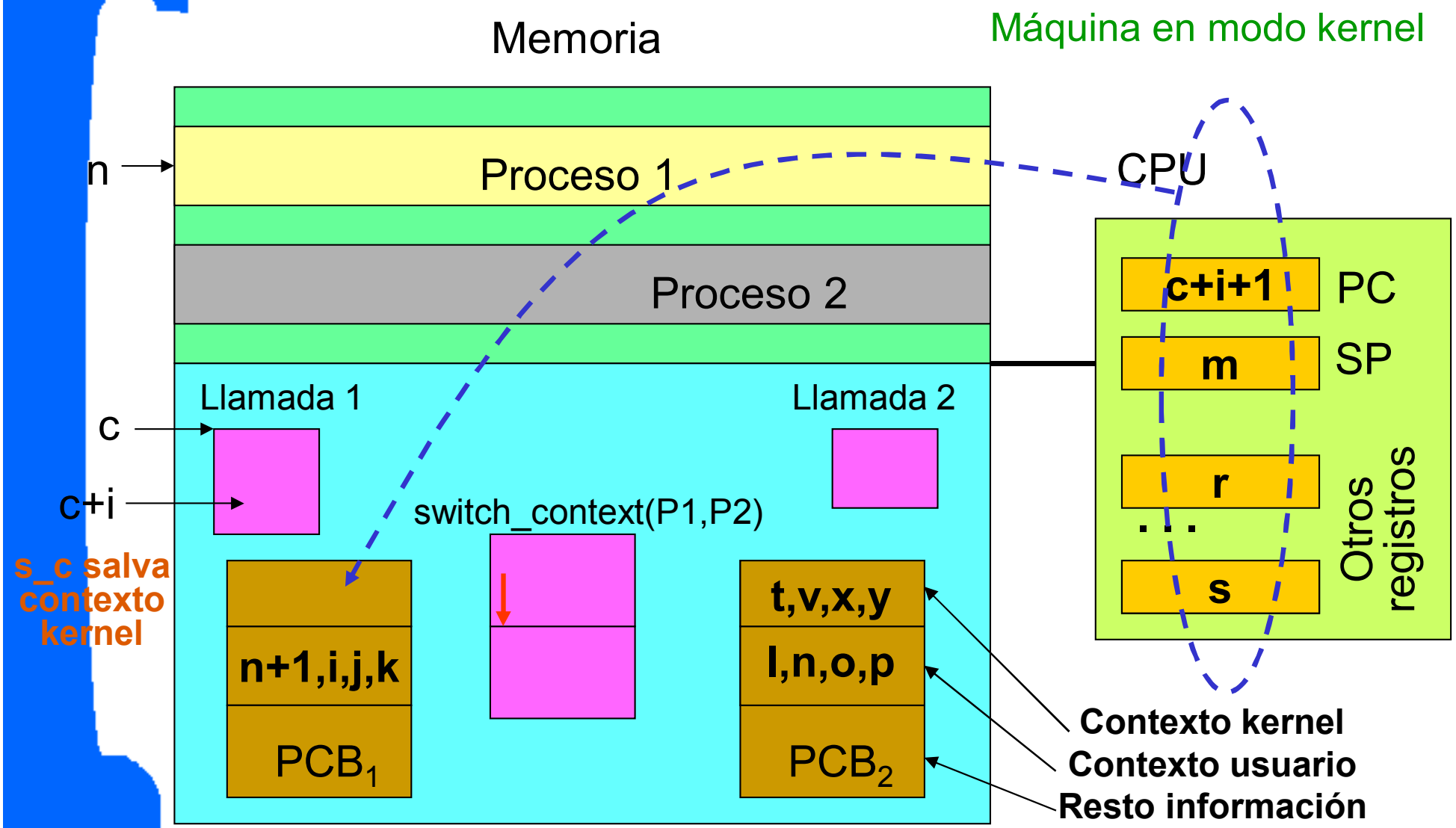
2º - Salva contexto usuario y ejecuta f^{on} kernel



3º - Parar proceso, invoca a `cambio_contexto`



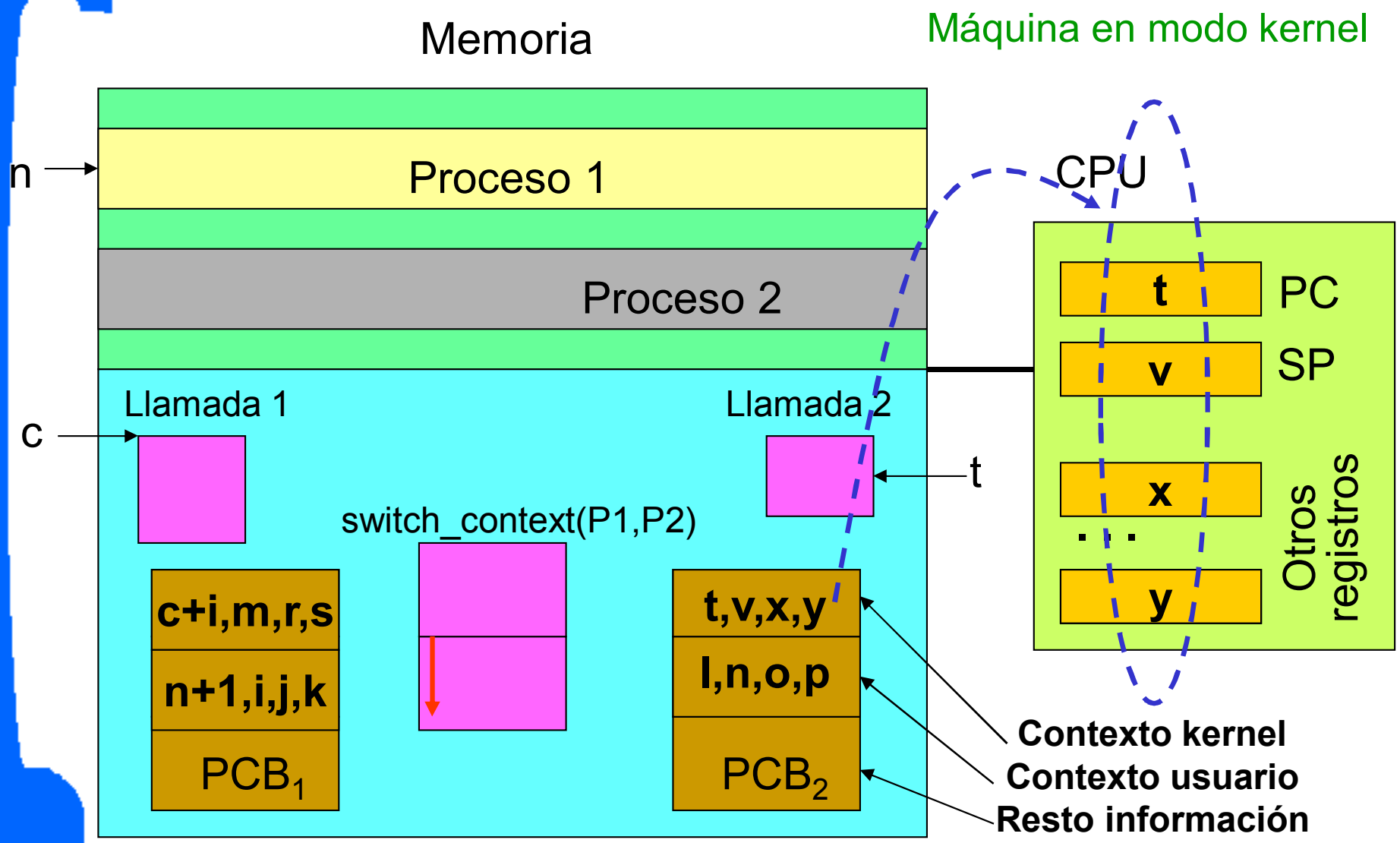
4º - Cambio_contexto()
salva contexto kernel



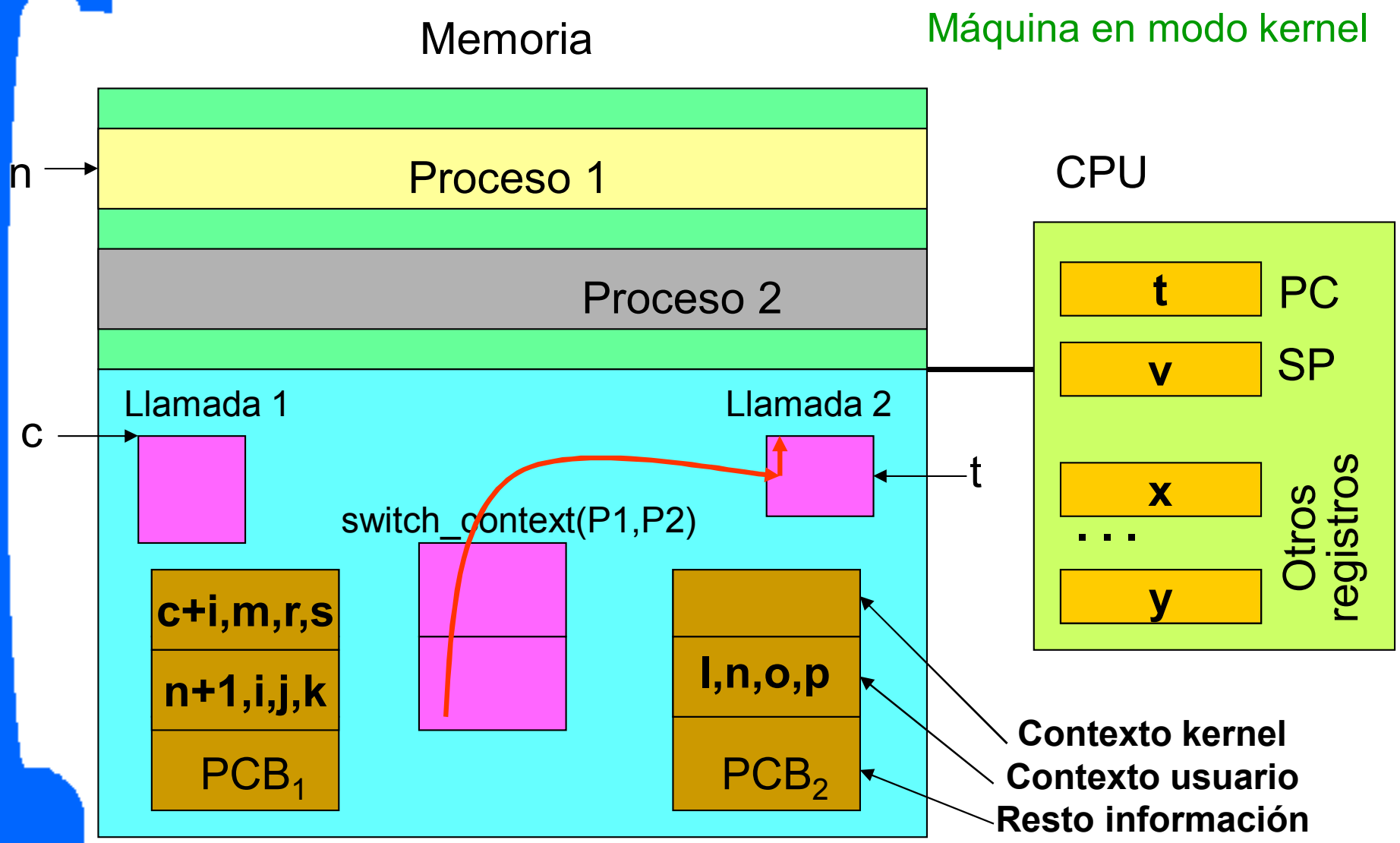
¿Cómo estamos?

- 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.

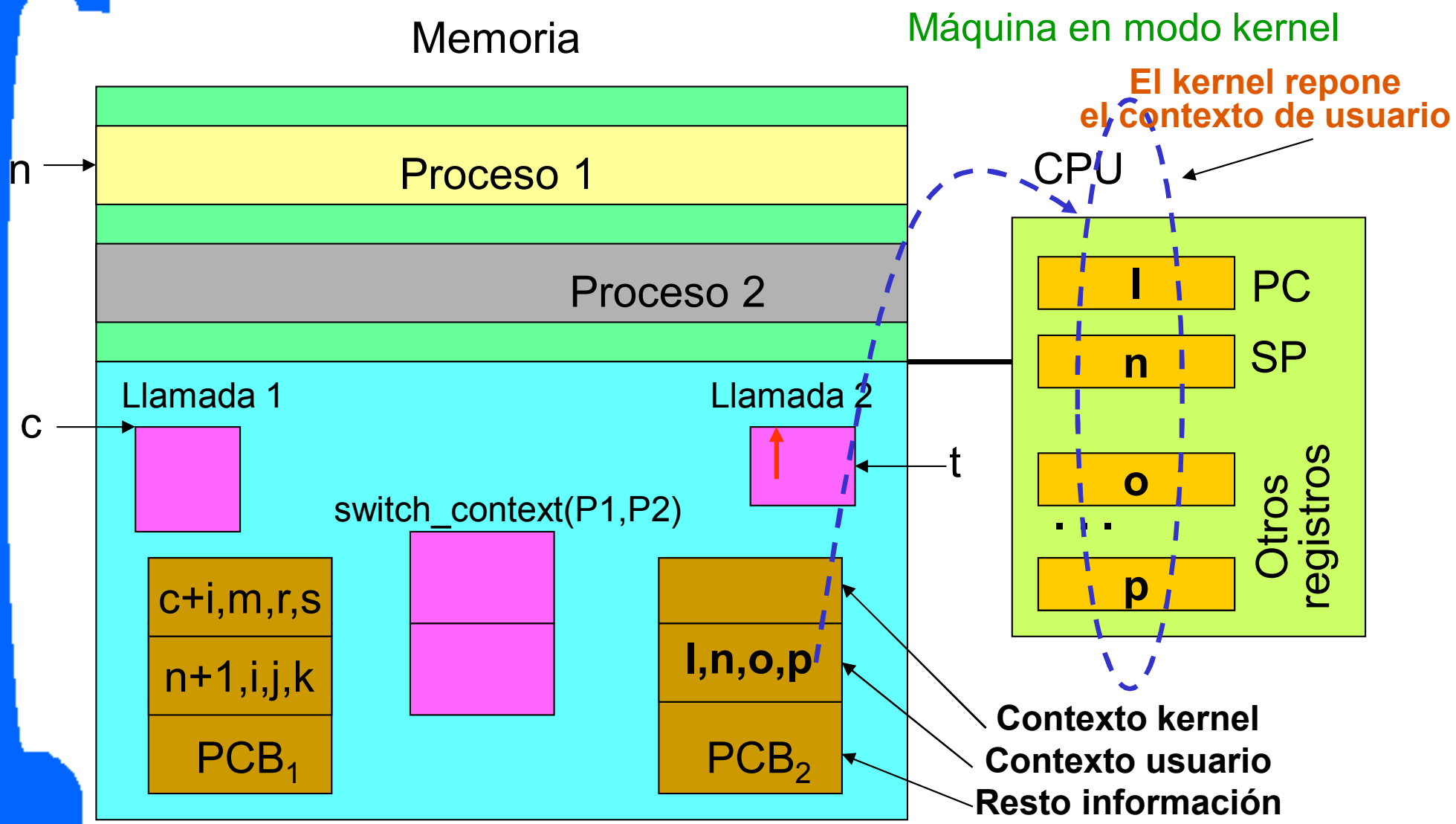
5º - Repone contexto kernel de P_2



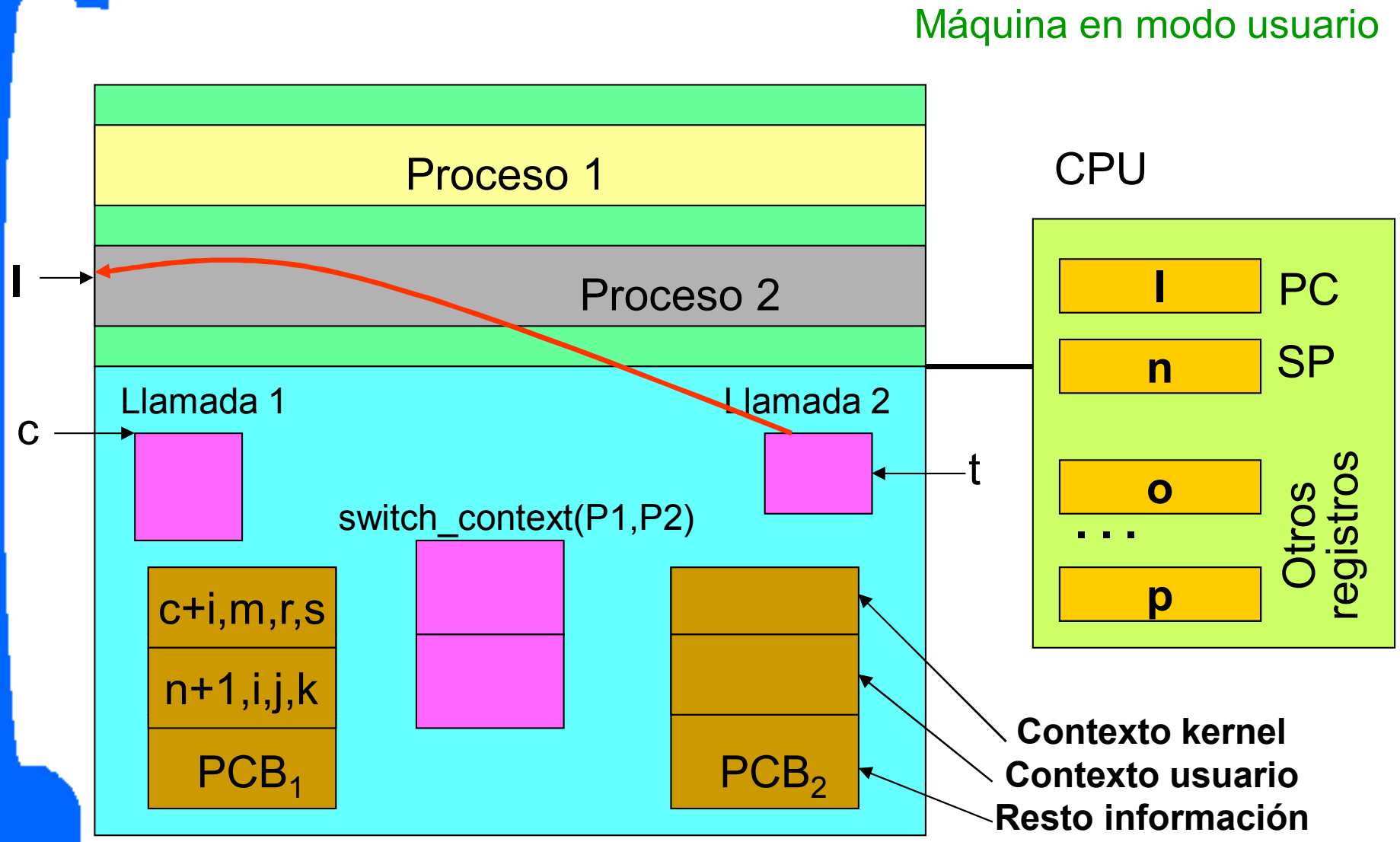
6º - El kernel termina la fon que inicio de P_2



7º - Finalizada fon, retorna a modo usuario



8º - reanudamos ejecución de P_2

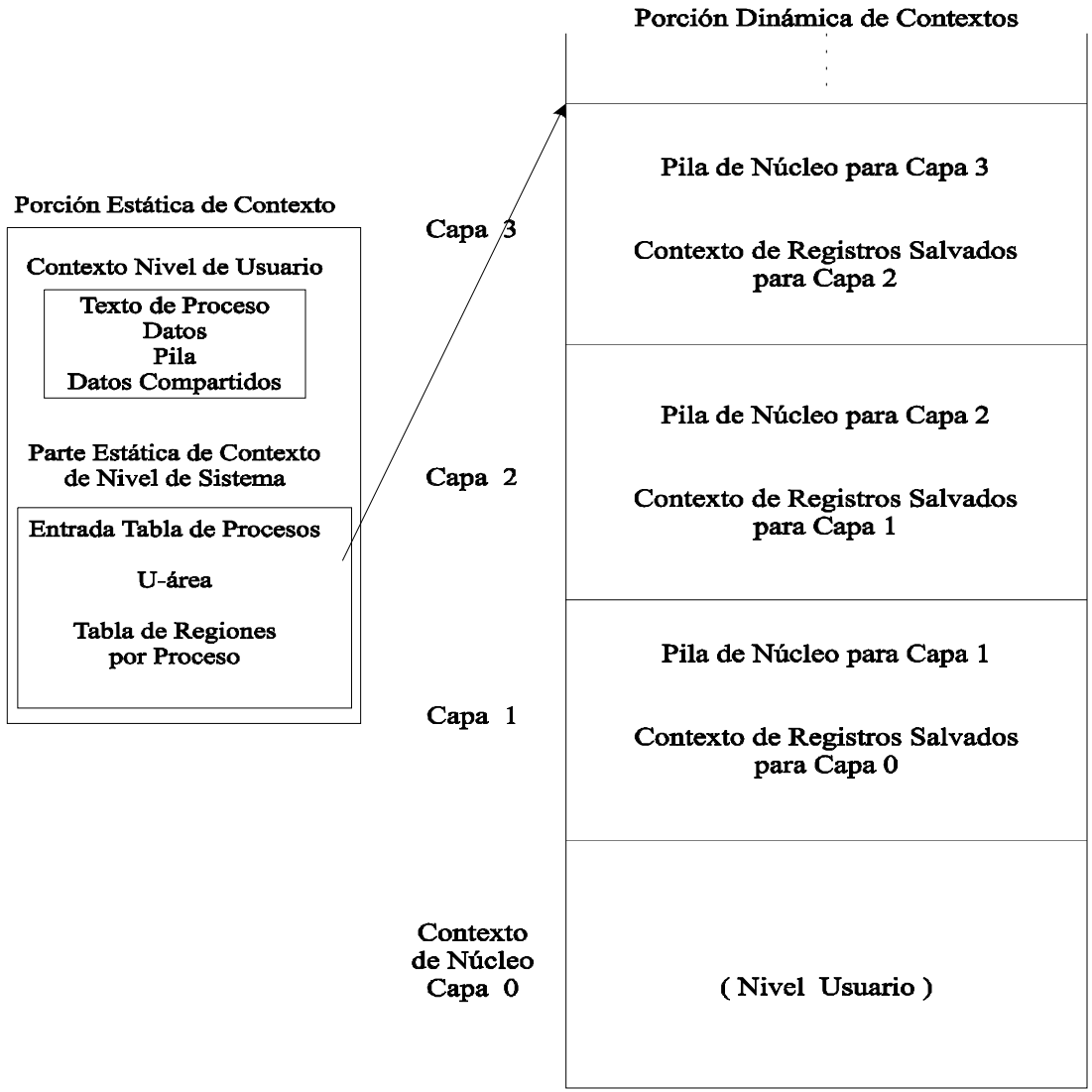


Contexto proceso

- Es el **estado** del proceso, la unión de:
  1. Contexto a nivel de **usuario** (texto, datos, pila)
  2. Contexto a nivel de **registros**
     - *PC*, dirección de la siguiente instrucción
     - Registro de estado del procesador (estado hardware)
     - *SP*, puntero de pila
     - Registros de propósito general
  3. Contexto a nivel de **sistema**
     - Parte estática, fija (entrada TP, u-área, pregion)
     - Parte dinámica, variable (pila de capas de contexto)

Contexto proceso (II)

- Se introduce una capa:
  - ocurre una interrupción
  - llamada al sistema
  - cambio de contexto
- Se saca una capa:
  - regreso de interrupción
  - regreso a modo usuario
  - cambio de contexto
- Un proceso se ejecuta dentro de su capa de contexto actual



Observaciones

- Cuando conmutamos al proceso P_2, este tiene la estructura de PCB que aparece el 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 *CrearProceso()* 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.
  - Se crea un contexto kernel para que parezca que el proceso retorna de una llamada al sistema.

Nueva pregunta ¿por qué hacer esto así? ...

Consistencia estructuras

- Es un problema de **Exclusión Mutua**
- En Unix tradicional
  - **enmascaramiento** de interrupciones
  - núcleo **no apropiativo**: los procesos que se ejecutan en modo supervisor no pueden ser apropiados
  - **bloquear** a los procesos cuando una estructura del núcleo está en uso por otro proceso (*sleep/wakeup*)
- Los núcleos actuales son apropiativos (total como Solaris, o parcialmente como SVR4)

Tipos de procesos

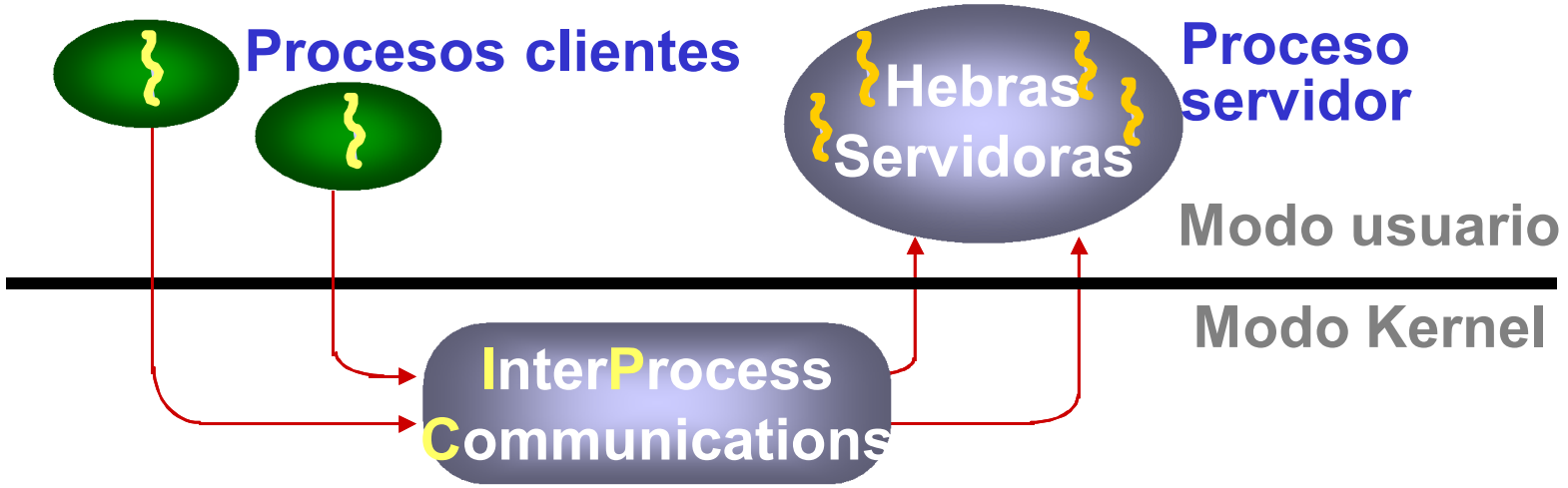
- Procesos de **usuario**
- Procesos **demonios** (*daemons*)
  - No están asociados a ningún terminal
  - Realizan funciones del sistema (spooler de impresión, administración y control de redes, ...)
  - Pueden crearlos el proceso **Init** o los procesos de usuario
  - Se ejecutan en modo usuario
- Procesos del **sistema**
  - Se ejecutan en modo supervisor
  - Los crea el proceso 0
  - Proporcionan servicios generales del sistema
  - No son tan flexibles como los demonios (recompilación)

Tipos de procesos

- El núcleo identifica a los procesos por su **PID**
- En Unix los procesos se crean con la llamada al sistema *fork* (excepto el proceso 0)
- Procesos especiales:
  - **Proceso 0**: creado "a mano" cuando arranca el sistema. Crea al proceso 1 y se convierte en el proceso **intercambiador**
  - **Proceso 1** (*Init*): antecesor de cualquier proceso del sistema

Limitaciones de procesos

1. Por naturaleza paralela de aplicaciones necesitan un espacio común de direcciones, p.e. servidores idempotentes.

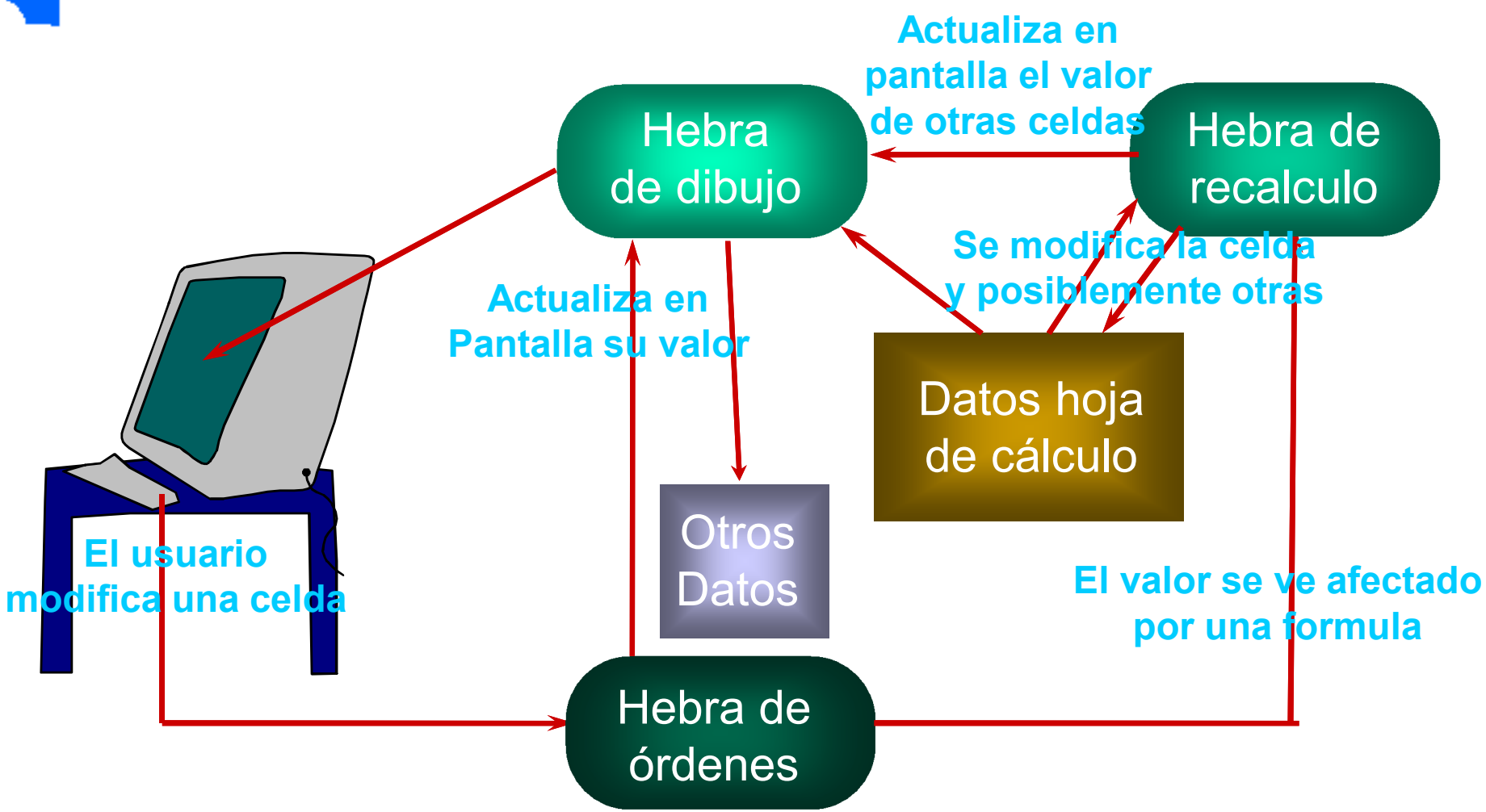


2. Un proceso sólo puede usar un procesador a la vez.

Hebras (*Threads*)

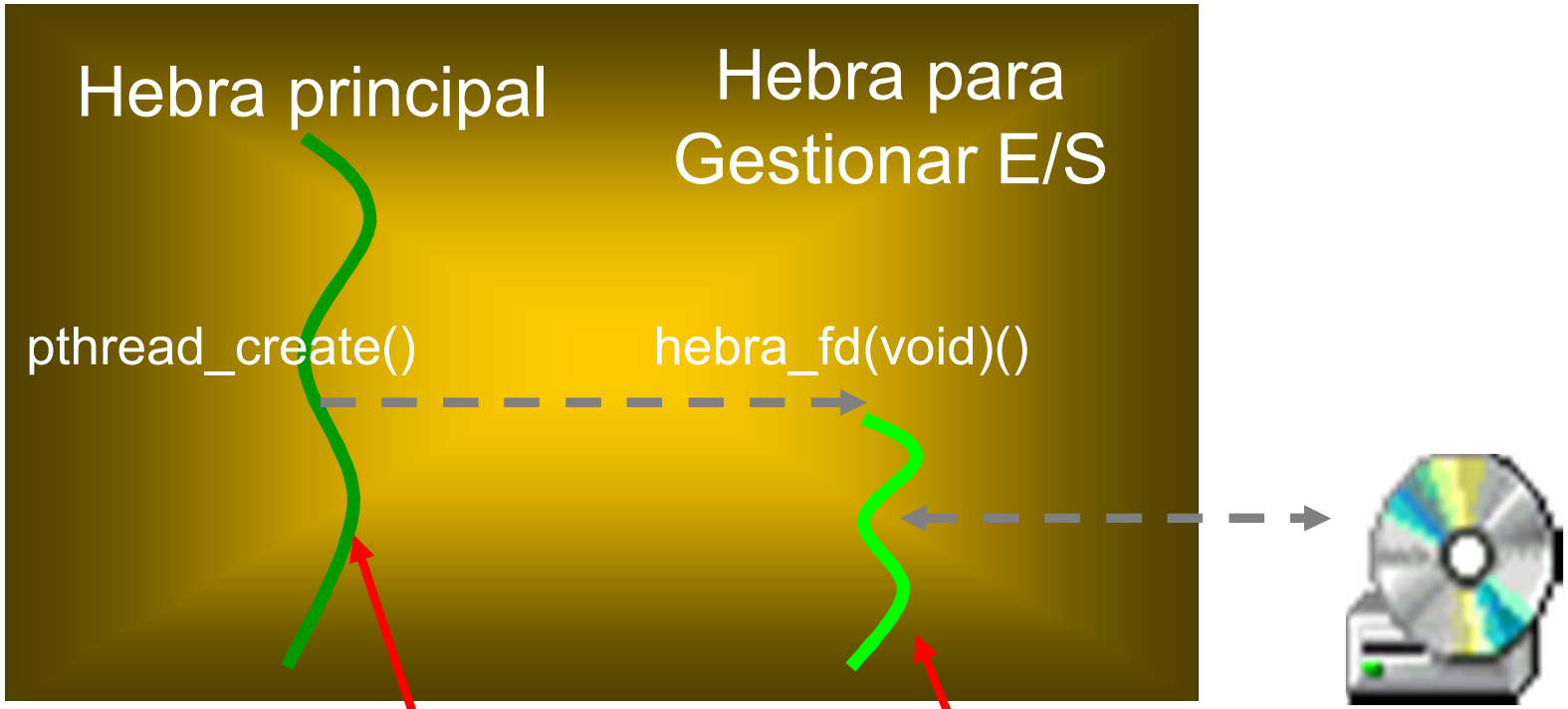
- En un proceso confluyen dos ideas que podemos separar:
  - **flujo de control** – secuencia de instrucciones a ejecutar determinadas por PC, la pila y los registros
  - **espacio de direcciones** – direcciones de memoria y recursos asignados (archivos, ...)
- Permitir más de un flujo de control dentro del mismo espacio de direcciones

Hoja de cálculo multihebrada



E/S asíncronas

Proceso multihebrado



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

¿Qué programas multihebrar?

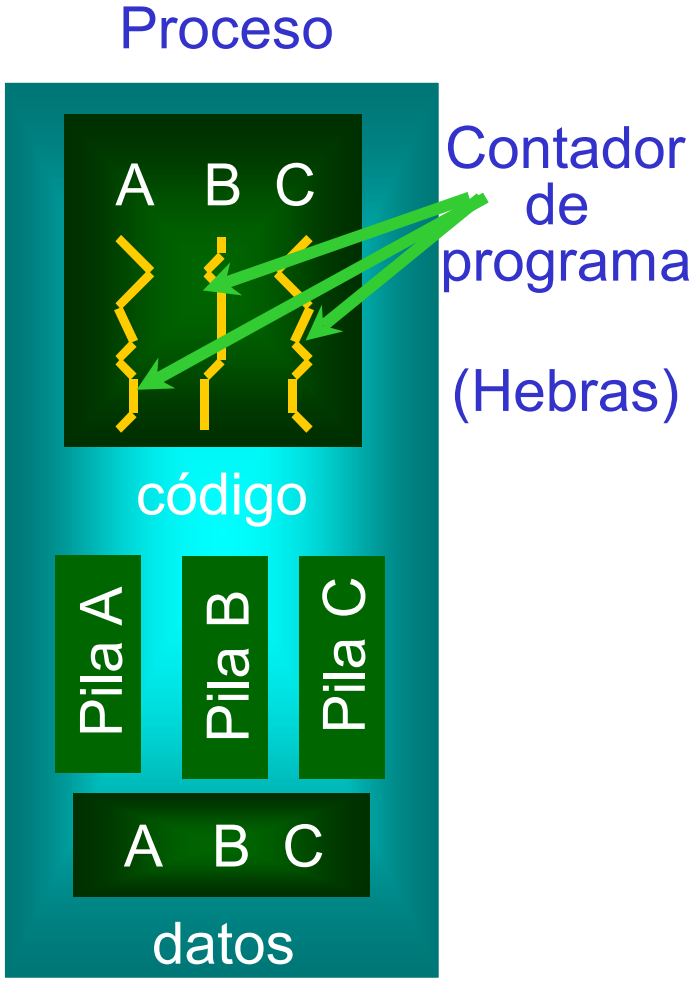
- ✓ Tareas independientes:depurador necesita GUI, E/S asíncronas, ...
- ✓ Programas únicos, operaciones concurrentes: servidores de archivos y web, Kernels de SOs: ejecución simultánea de varios peticiones de usuario, ...
- ✓ Uso del hardware multiprocesador
- ✘ Si todas las operaciones son CPU intensivas
- ✘ La hebras tienen algún coste aunque menor que los procesos (e.g. memoria): una hebra con pocas líneas de código no es muy **útil**

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 en el módulo o estáticos.
- El SO debe ser código reentrante.
- P.ej. El kernel 2.4 Linux no es 100% reentrante, el 2.6 sí. MS-DOS y la BIOS no son reentrantes.

Diseño

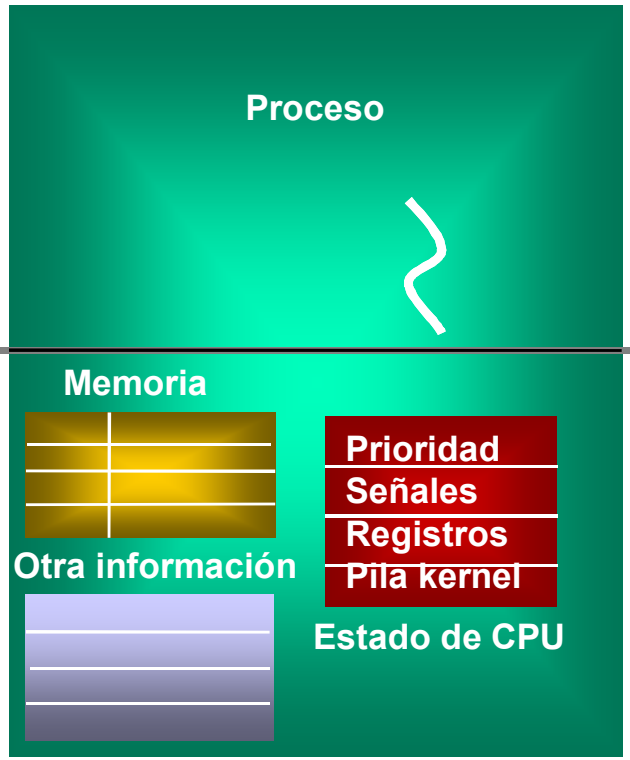
- Hebra= unidad de asignación de la CPU (de planificación).
- Tiene su propio contexto hardware:
  - Su valor del contador de programa.
  - Los valores de los registros.
  - Su pila de ejecución.



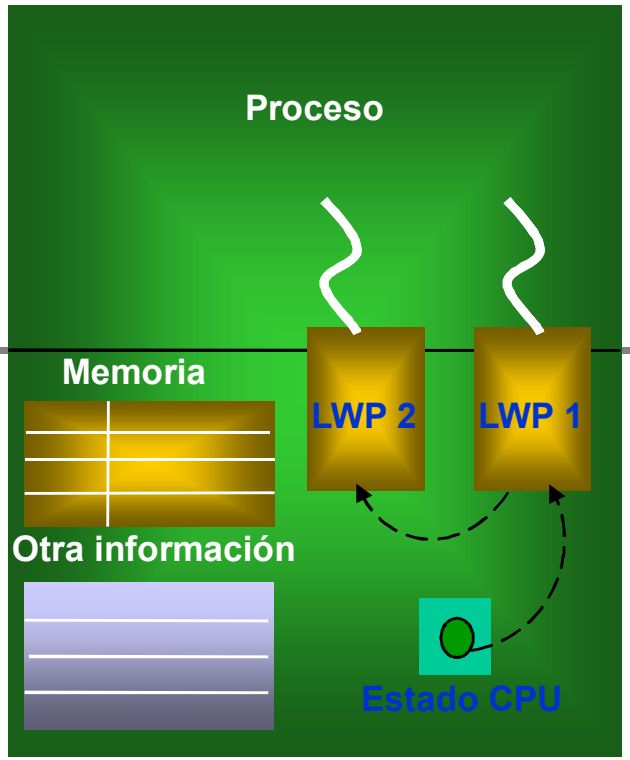
“ Programa multihebrado”

Implementación de LWP

Estructura de un proceso Unix tradicional



Estructura de proceso en Solaris 2



Tipos de hebras

- Hebras Kernel – implementadas dentro del kernel. Conmutación entre hebras rápida.
- Hebras de usuario – implementandas 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.

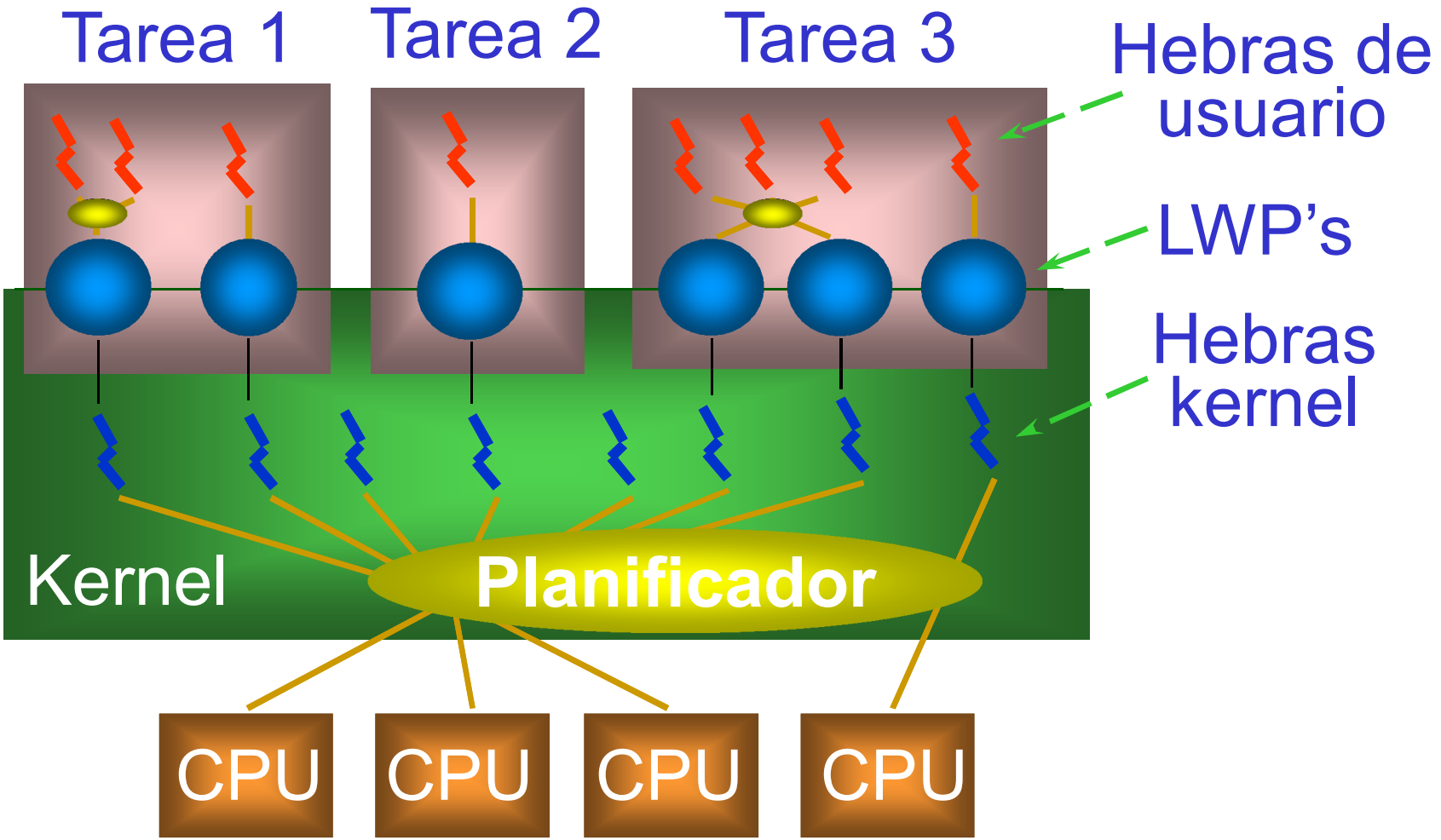
Estándares de hebras

- 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.
- Hebras Java

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.

Tareas, hebras, y LWP's



Hebras en Linux

- Linux soporta:
  - Hebras kernel en labores de sistemas – ejecutan una única función en modo kernel: 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

APIs de Unix y Win32

Operación	Unix	Win32
Crear	`fork()` `exec()`	`CreateProcess()`
Terminar	`_exit()`	`ExitProcess()`
Obtener código finalización	`wait` `waitpid`	`GetExitCodeProcess`

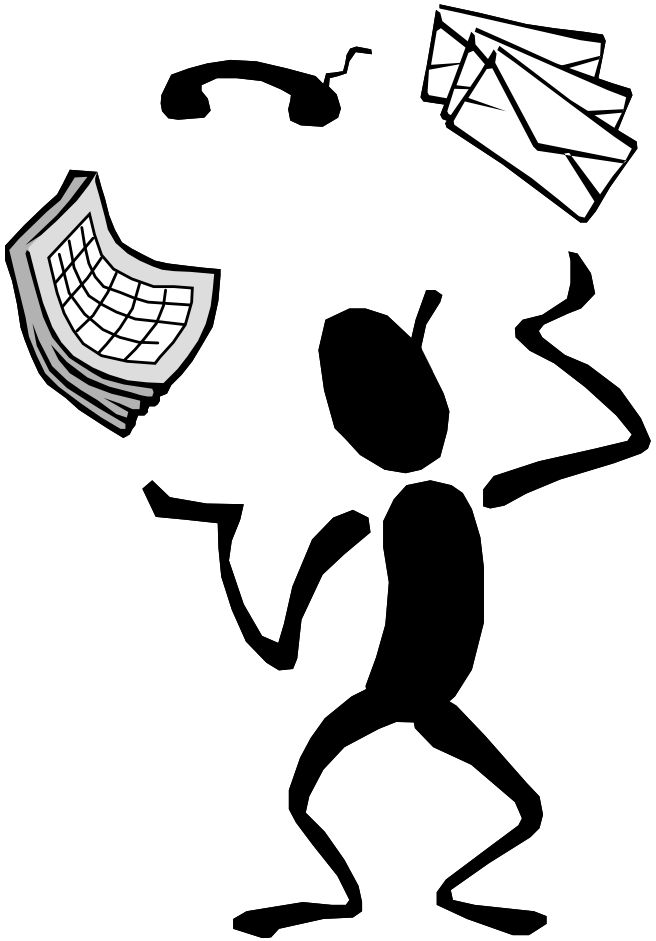
Obtener tiempos	`times`
	`wait3`
	`wait4`
		`GetProcessTimes`

Identificador	`getpid`	`GetCurrentProcessId`
Terminar otro proceso	`kill`	`TerminateProcess`

APIs de hebras

Operación	Pthread	Win32
Crear	`Pthread_create`	`CreateThread`
Crear en otro proceso	-	`CreateRemoteThread`
Terminar	`Pthread_exit`	`ExitThread`
Código finalización	`Pthread_yield`	`GetExitCodeThreaad`
Terminar	`Pthread_cancel`	`TerminateThread`
Identificador	-	`GetCurrentThreadId`

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.
- Planificador a corto plazo (o *de la CPU*) – selecciona al proceso que debe ejecutarse a continuación, y le asigna la CPU.

Características 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 se lento.
- Controla el grado de multiprogramación (nº 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.
- 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 :
  1. Estado ejecutándose a bloqueando.
  2. Estado ejecutándose a preparado.
  3. Estado bloqueado a preparado.
  4. 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:
  1. Cambio de contexto (en modo kernel).
  2. Conmutación a modo usuario.
  3. 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 seudocódigo:

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

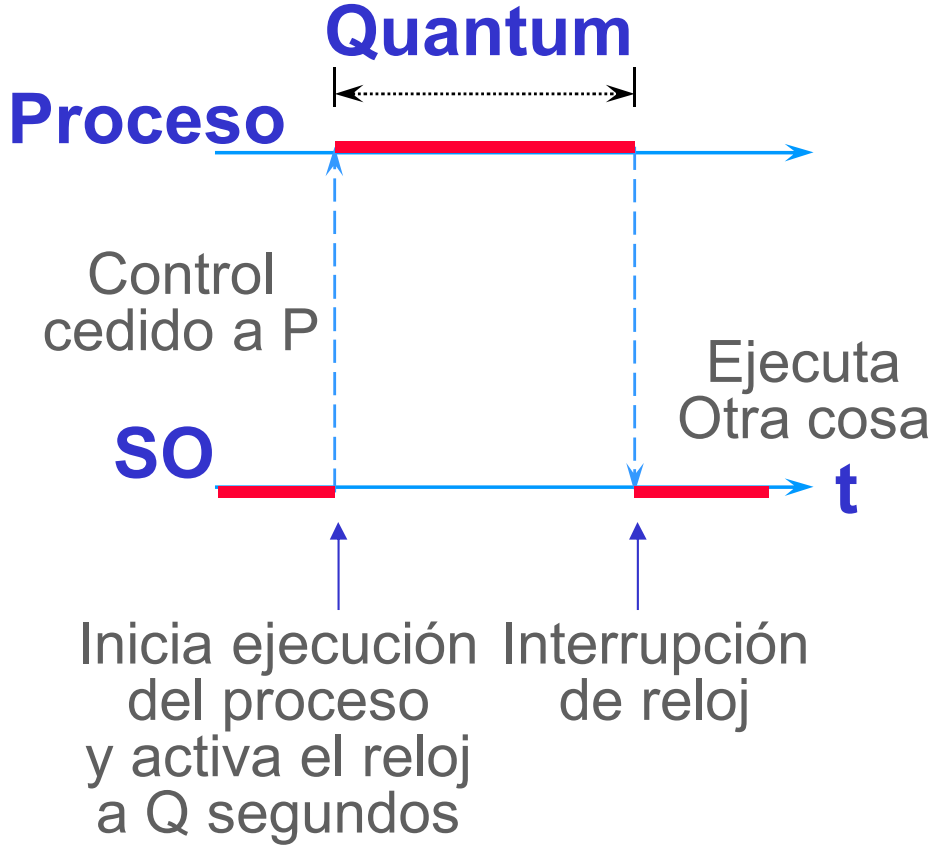
- ¿Cómo obtiene el despachador el control ?
  - Síncrona – un proceso cede la CPU.
  - Asíncrona – iniciado por una interrupción u ocurrencia de un evento que afecta a un proceso (p. ej. fin de e/s, liberación recurso,...)

Gestor interrupciones revisado

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

Implementación 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 vs. 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

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

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

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

Primero en Llegar, Primero en Servir (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):
  $(0+24+27)/3 = 17$

a)

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.

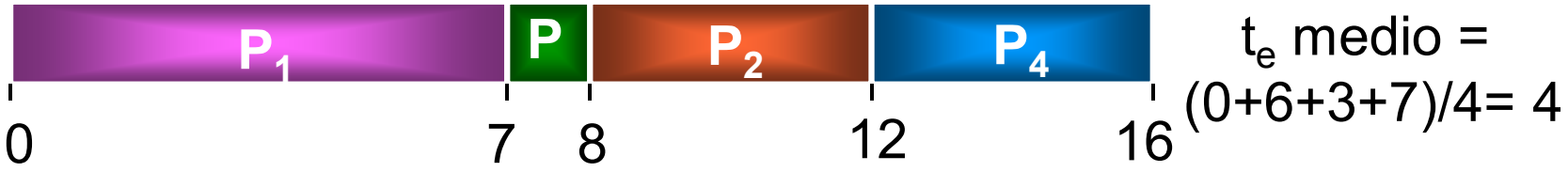
Características de SJF

- 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

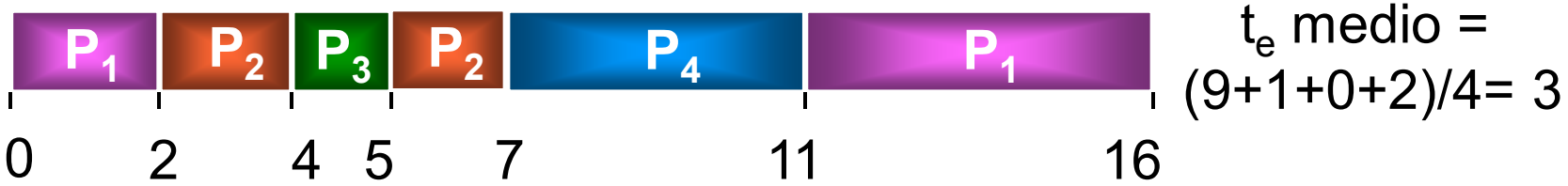
Proceso	T. Ráfaga	T. llegada
P_1	7	0
P_2	4	2
P_3	1	4
P_4	4	5

- SJF (no apropiativo)



t_e medio = (0+6+3+7)/4= 4

- SRTF (apropiativo)



t_e medio = (9+1+0+2)/4= 3

Estimación tiempo de 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, asocia-mos 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 60–120 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.
- Se mostraba con se controla el tiempo del quantum mediante el reloj.

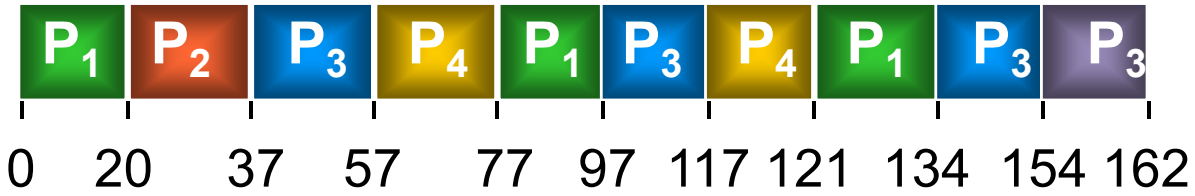
Características de RR

- 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*.

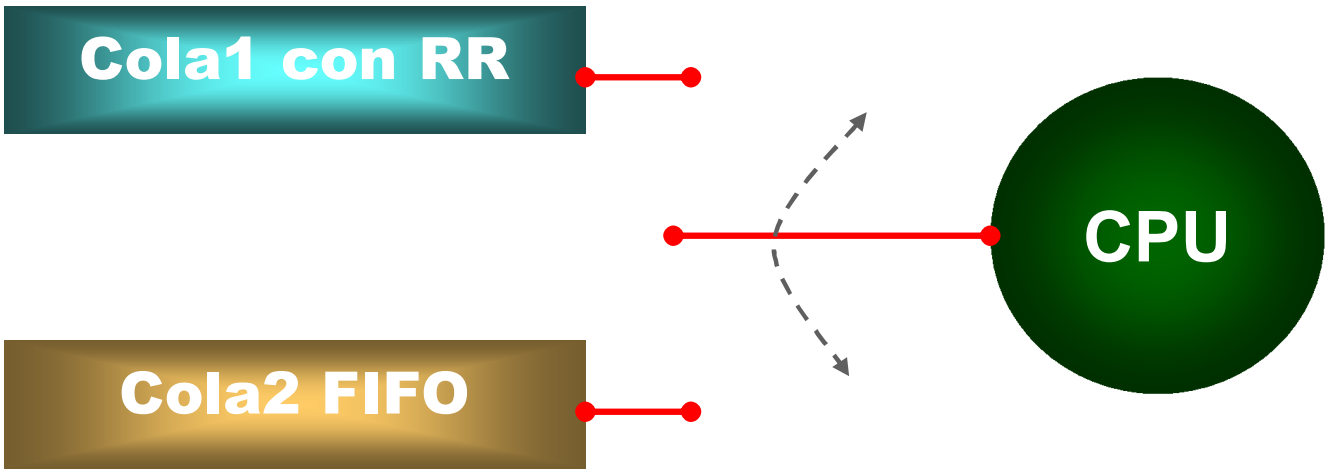
Colas múltiples

- 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

2. 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.

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 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 Tiempo-Real (II)

- Es importante reducir la latencia de despacho para acotar el tiempo de respuesta:
- Como algoritmos de planificación se suelen utilizan 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.

Inversión de prioridad

- Fenómeno producido cuando un proceso o hebra, P_2, de prioridad p_2, se ejecuta antes que P_1, de prioridad $p_1 > p_2$, debido a que P_1 espera por recurso que tiene bloqueado P_3.
- Una solución: herencia de prioridad, P_1 lega su prioridad a P_3 (que bloquea el recurso)

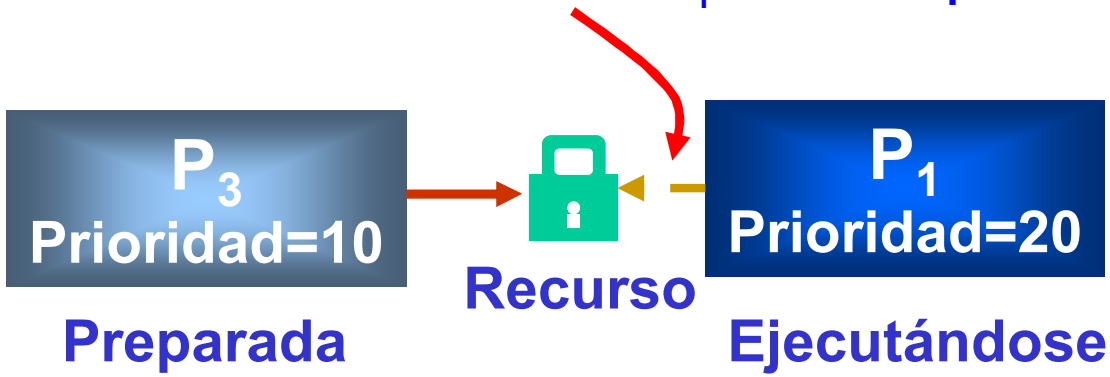
Inversión de prioridad (II)

Sea P_1 que se desbloquea o se crea ->
P_1 se ejecuta por tener mayor prioridad

Ejecutánse ->Preparada **Preparada –>Ejecutarse**

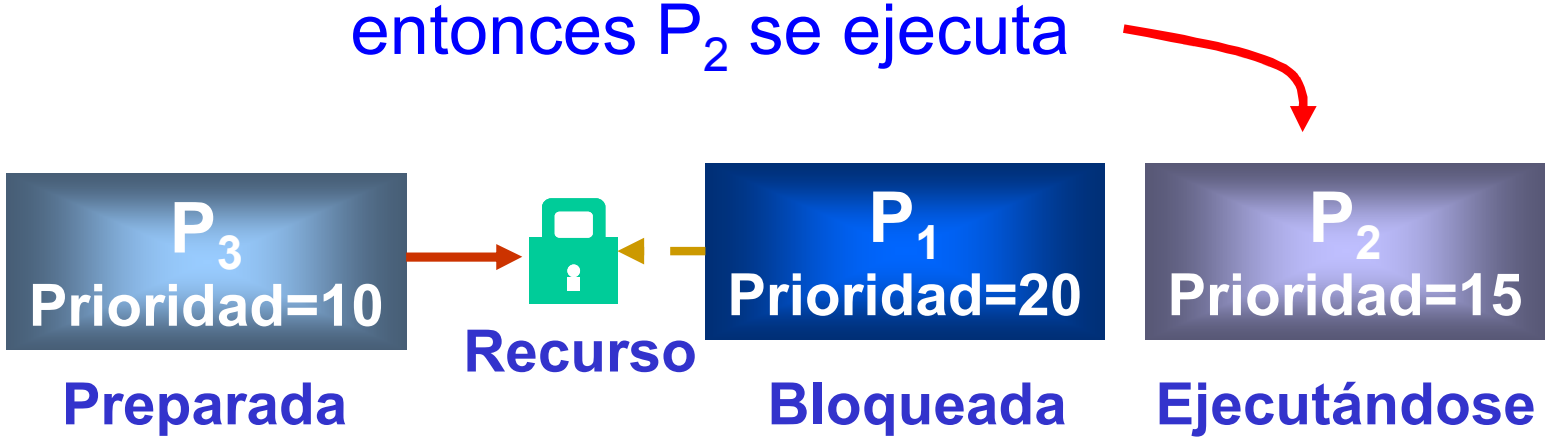
Inversión de prioridad (III)

P_1 solicitar recurso: si no se puede apropiar P_3, entonces P_1 se bloquea



Inversión de prioridad (IV)

Si aparece un P_2 con prioridad mayor que P_3, entonces P_2 se ejecuta



Control de procesos

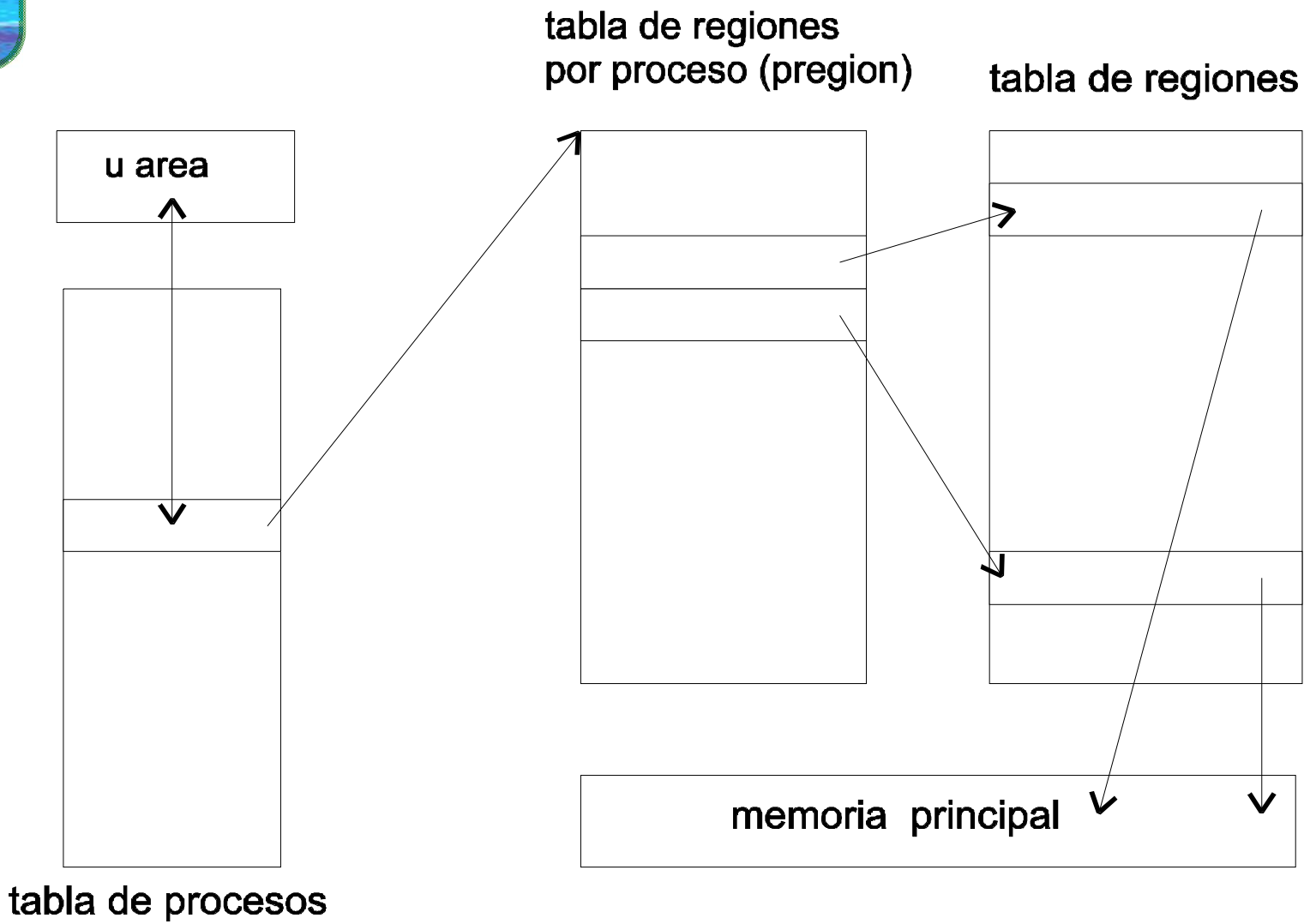
- Llamadas al sistema para procesos:

Relacionadas con Gestión de Memoria				Relacionadas con sincronización		
fork	*exec*	*brk*	*exit*	*wait*	*signal*	*kill*
algoritmos internos (*sleep, wakeup, ...*)						

Estructuras de datos

- Cada proceso tiene una **entrada en la tabla de procesos**: contiene información de control y estado sobre el proceso
- Cada proceso tiene una **tabla de regiones privada** (*pregión*) cuyas entradas apuntan a entradas de una **tabla de regiones global**

Estructuras de datos



¿Objetivo la división de la tabla de regiones (global y otra específica por proceso) ?

Tabla de procesos

- **Estado** del proceso
- Campos para localizar el proceso y su u-área, e información sobre el tamaño del proceso
- Relaciones entre procesos
- Descriptor de eventos que espera un proceso
- Parámetros de planificación

Tabla de procesos

- Señales recibidas pero no tratadas aún
- Temporizadores indicando el tº de ejecución del proceso y el tº de uso de los recursos
- Punteros para enlazar el proceso en la cola del planificador, o si está bloqueado, en la cola de bloqueados
- Punteros para las colas hash en base a su PID

Tabla de procesos o U-área

- Puntero a la entrada de la tabla de procesos
- UIDs y GIDs real y efectivo
- Tiempos que los procesos y sus descendientes gastan ejecutándose en modo usuario y modo supervisor
- Matriz que almacena la reacción ante las señales
- Terminal asociado al proceso, si existe
- Campos de error y del valor devuelto (para las llamadas al sistema)

Tabla de procesos o U-área

- Tabla de descriptores de archivos
- Parámetros de las operaciones de E/S
- Directorio actual y raíz
- Tamaños límites de procesos y archivos
- Campo modo de permisos
- Pila del núcleo

pregion

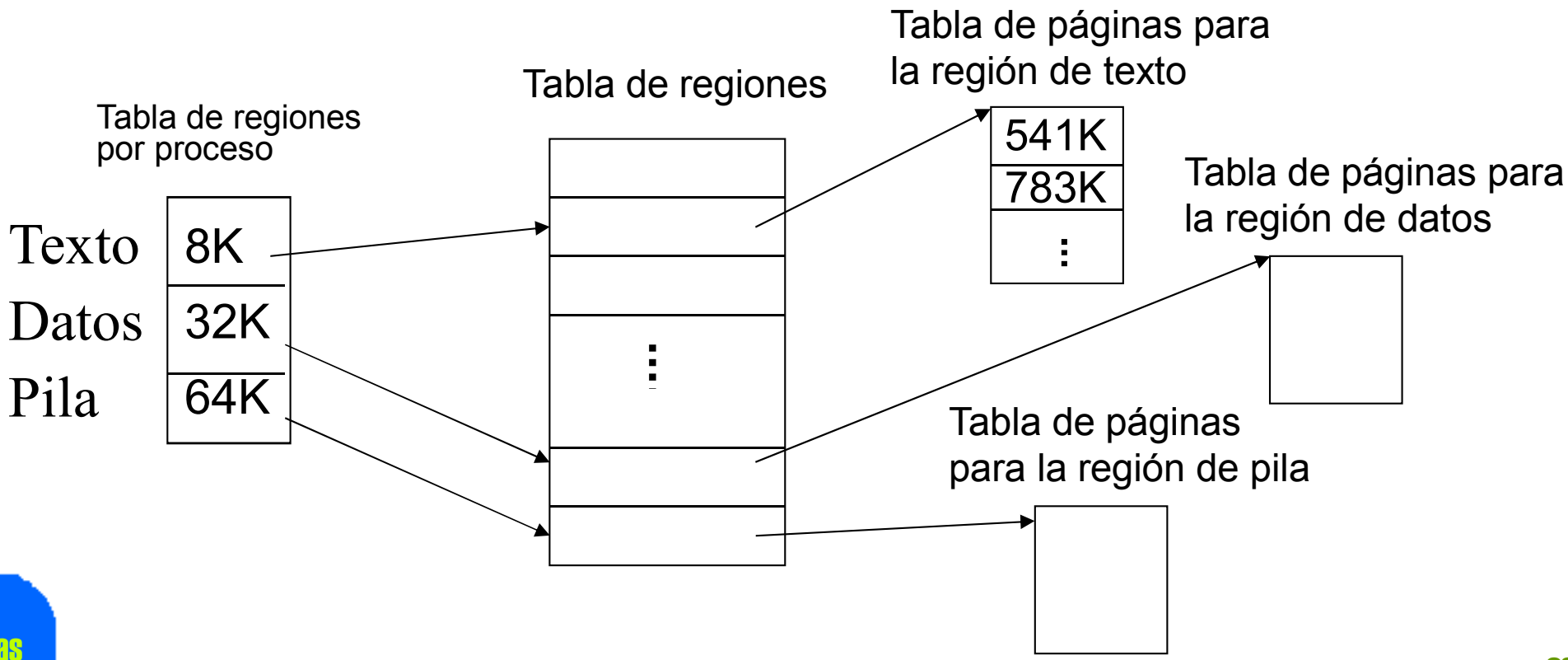
- Puede estar en distintos lugares, depende de la implementación:
  - en la tabla de procesos o u-área
  - en una zona de memoria asignada separadamente
- Contenido de cada entrada:
  - puntero a la correspondiente entrada de la tabla de regiones
  - dirección virtual de comienzo de la región
  - campos de permiso para el tipo de acceso del proceso
- El concepto de región es independiente de las políticas de gestión de memoria utilizadas

Tabla de regiones

- Puntero al **i-nodo del archivo ejecutable** desde donde se cargaron originalmente sus contenidos
- **Tipo de región**: texto, datos compartidos, pila, ...
- **Tamaño** de la región
- **Localización** de la región en memoria principal
- **Estado** de la región (bloqueada, en demanda, cargándose en memoria o válida)
- **Contador** de procesos que referencian la región

Diseño de Memoria

- Se utiliza un sistema de paginación por demanda: cada región tiene asociada una tabla de paginas con información de las páginas que la forman



Creación de procesos

- Los pasos a seguir por el SO:
  1. Asignarle un PCB
  2. Establecer su contexto de memoria (espacio de direcciones)
  3. Cargar imagen (ejecutable) en memoria
  4. Ajustar su contexto de CPU (registros)
  5. 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 (II)

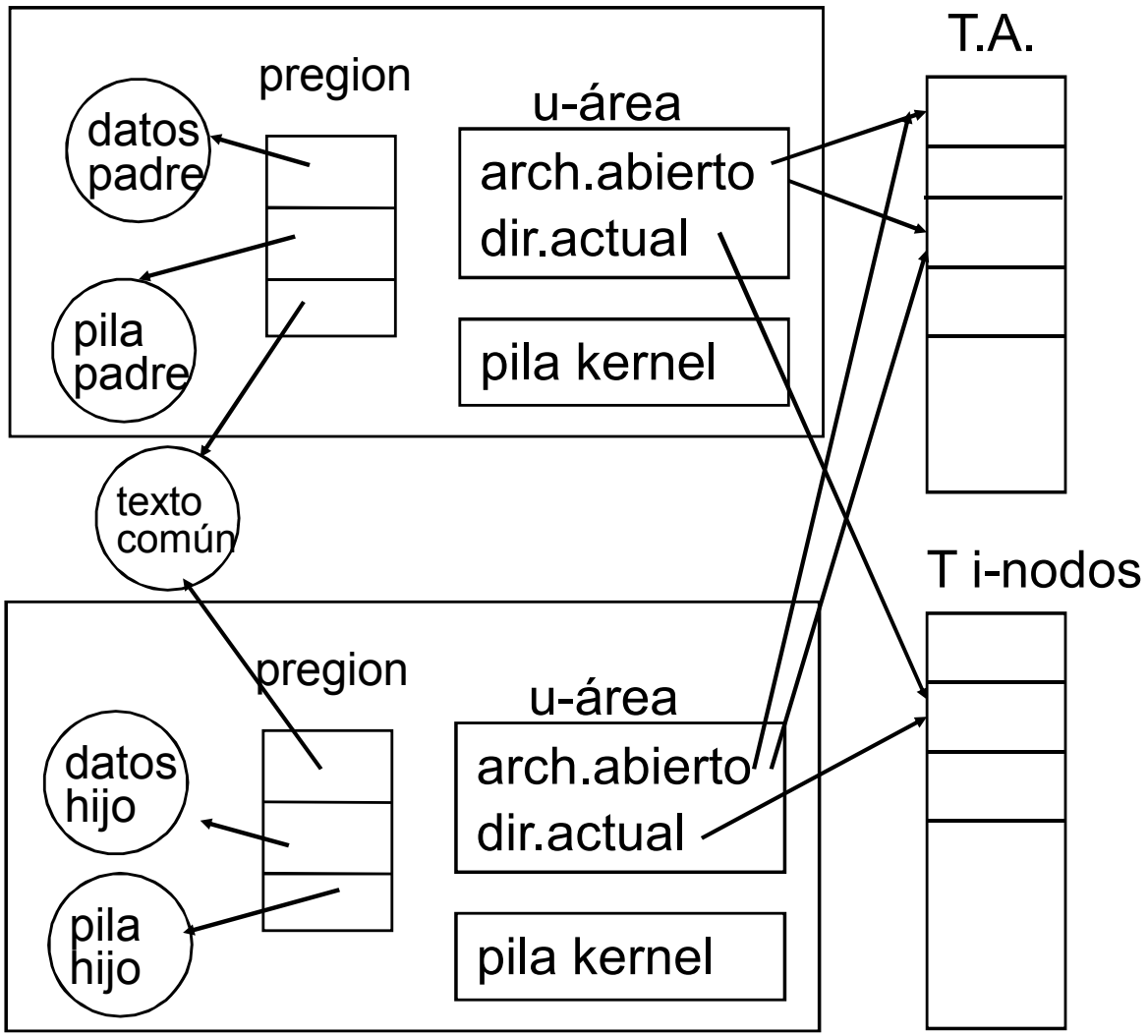
- Respecto a la ejecución:
  - Creador/creado se ejecutan concurrentemente.
  - Creador espera al que el creado termine.
- Sus espacios de direcciones son:
  - Clonados –se copia el espacio del creador para el creado: comparten el mismo código, datos, ... ej. Unix.
  - Nuevos –el proceso creado inicia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

Llamada al sistema *fork*

- Crea un nuevo proceso, copia casi idéntica del padre
- Devuelve al padre el PID del hijo y al hijo 0
- Sintaxis:

```
pid = fork()
```

- Después del fork, ambos se ejecutan concurrentemente



Llamada al sistema *exec*

- Ejecuta el programa que se le pasa como argumento
- Existen seis funciones *exec* que se diferencian únicamente en la forma de pasar los argumentos (*execl*, *execv*, *execvp*, ...)
- Si la llamada tiene éxito, el espacio de direcciones del proceso que la inició se ha sustituido completamente por un nuevo programa
- Hay que asegurar que el nuevo programa se ejecutará correctamente. P.e. puede ser necesario redirigir la entrada o la salida o ambas

Llamadas *exit* y *wait*

- ***exit***
  - Pone fin a la ejecución de un proceso
  - Devuelve el estado de finalización al padre y el proceso pasa a estado *zombie*
- ***wait***
  - Espera de la teminación del primer hijo
  - Si no ha terminado ningún hijo, el proceso (padre) se bloquea
- Si un padre finaliza antes que sus hijos, los hijos se conectan al proceso *Init*
- Si un hijo finaliza sin que el padre ejecute wait, se queda en estado *zombie*

Manejador del reloj

- Toda máquina Unix tiene un reloj hardware que interrumpe al sistema en intervalos de tº fijos (típicamente se establece el *tick* de reloj en 10ms)
- Las funciones del manejador de la interrupción de reloj son (no todas en cada tick):
  - Reiniciar el reloj (si es necesario)
  - Planificar funciones internas del núcleo basadas en temporizadores
  - Reunir estadísticas del sistema y de los procesos
  - Mantener la hora
  - Enviar las señales de alarma a los procesos
  - Controlar la planificación de procesos
  - Despertar al *intercambiador* y *stealer* periódicamente

Planificación Unix

- Cada proceso activo tiene una prioridad de planificación
- El planificador utiliza la política de **colas múltiples con realimentación**, donde cada cola se gestiona por **Round Robin**
- El **algoritmo** que se sigue es:
  - selecciona el proceso de más alta prioridad de aquellos que están en estado apropiado o preparado para ejecutarse en memoria
  - si existe más de un proceso, elige el más antiguo
  - si no hay procesos, espera a la siguiente interrupción y después de su tratamiento, intenta planificar un proceso

Planificador SVR4

- Se introducen **clases de planificación**: definen la política de planificación para los procesos. Hay dos:

  1) **Tiempo compartido** (0 – 59):

  - Cambia la prioridad de un proceso en respuesta a eventos específicos relacionados con ese proceso
  - Existe una tabla de parámetros del planificador que define cómo los distintos eventos cambian la prioridad de un proceso
  - El quantum depende de la prioridad

Planificador SVR4 (II)

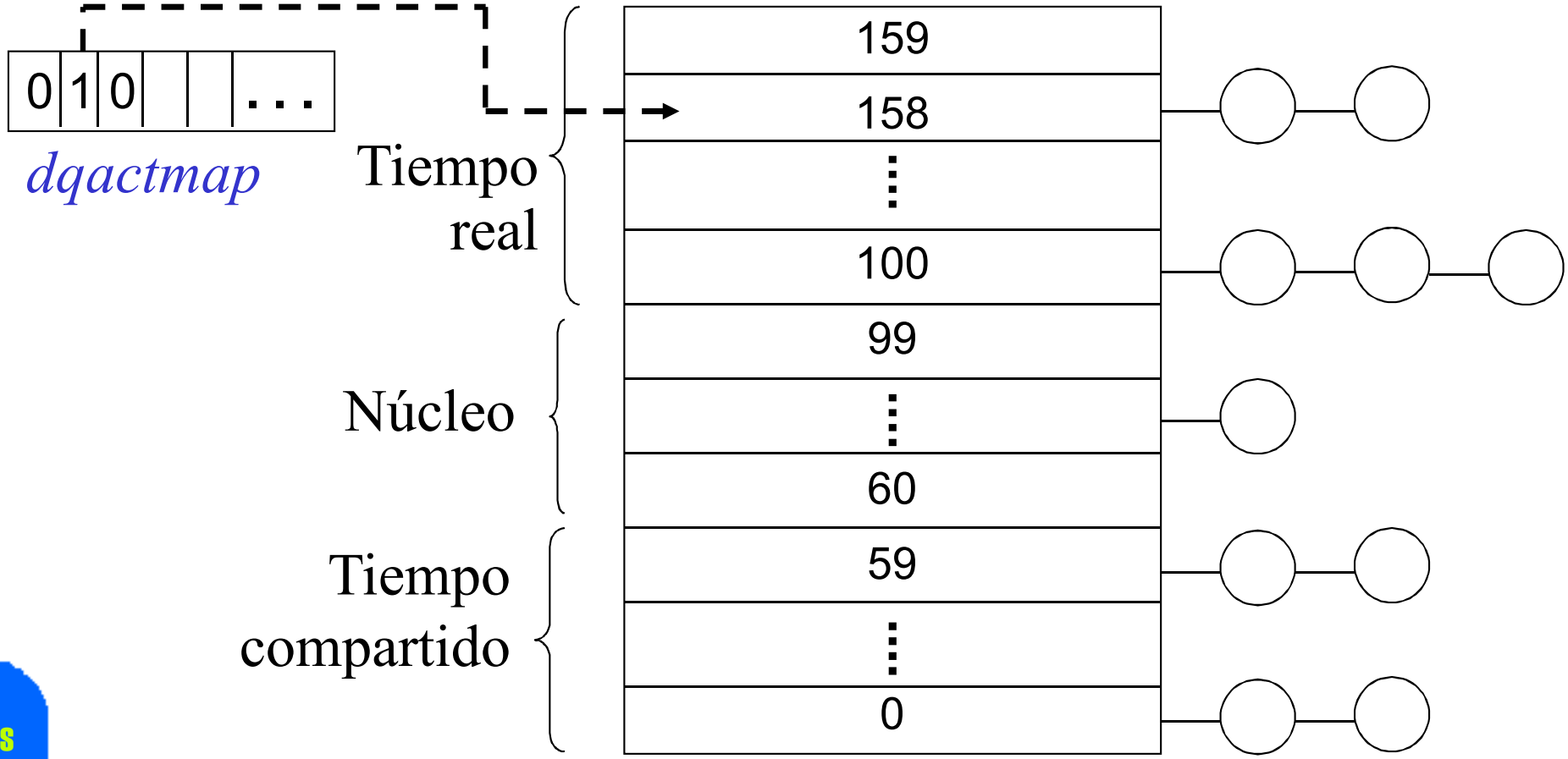
- Disminuye la prioridad cuando consume quantum y aumenta cuando se bloquea o tarda en usar la CPU
- Cuando un proceso se bloquea, se le asigna una prioridad a nivel de núcleo (60 – 99)
- Valor *nice* entre –20 a +19 (por defecto, 0)

2) Tiempo real (100 – 159)

- Los procesos tienen prioridad y quantum fijos especificados explícitamente
- Sólo el superusuario puede tener procesos en esta clase

Planificador SVR4 (III)

- Existe una interfaz para ciertas rutinas dependiendo de la clase (*calculoPrioridad*, ...)



Señales

- Informan a los procesos de la ocurrencia de eventos
- Los procesos y el núcleo pueden enviar señales
- Categorías

  1) Eventos síncronos: errores generados por la ejecución de un proceso. P.e. violación segmento (*SIGSEGV*) o instrucción ilegal (*SIGILL*)

  2) Eventos asíncronos: ocurren externamente a la ejecución del proceso pero que tienen alguna relación. P.e. terminación de un hijo (*SIGCHLD*) o bloqueo desde el manejador del terminal *tty* (*SIGHUP*)

Señales (II)

- Cada señal tiene asociado un número entero positivo
- Envío de señal a un proceso → el núcleo activa un bit del **campo de señales** en entrada tabla de procesos
- Un proceso puede bloquearse de dos formas:
  - **interrumpible**: el suceso que espera no se sabe cuando ocurrirá o si ocurrirá (ej. pulsación de una tecla). Si proceso está bloqueado de forma interrumpible → la recepción de señal provoca su desbloqueo
  - **no interrumpible**: el suceso ocurrirá pronto (ej. fin E/S)
- Un proceso puede recordar diferentes tipos de señales, pero no cuántas de cada tipo ha recibido

Señales (III)

- Hay dos fases en el proceso de señalización:

1) Generación: el núcleo genera señales en base a varios eventos:

  - Excepciones
  - Un proceso puede enviar una señal a otro u otros
  - Interrupciones del terminal
  - Control de tareas: el shell usa señales para manipular los procesos en *foreground* y *background*
  - Quotas: exceso en límites de CPU o tamaños de archivos
  - Notificaciones solicitadas por un proceso al sistema
  - Alarmas

Señales (IV)

2) Reparto o manejo: cuando el proceso ejecutándose va a pasar de modo supervisor a modo usuario comprueba señales pendientes. Posibles acciones:

- Abortar o terminar el proceso
- Ignorar
- Ejecutar una función concreta
- Suspender al proceso
- Reanudar un proceso suspendido

Cada señal tiene una acción por defecto que lleva a cabo el núcleo si no se especifica una alternativa.

- Hay señales especiales: SIGKILL o SIGSTOP

Señales (V)

- Para especificar la acción deseada:

  funcion_anterior = signal (num_señal, funcion)

  funcion puede ser:
  - dirección de la función a invocar
  - ignorar la ocurrencia de la señal
  - terminar el proceso

- Para enviar una señal: kill (pid, num_señal)

  Dependiendo del valor de pid, el núcleo envía la señal a:
  - pid > 0: al proceso con ese nº de pid
  - pid = 0: a todos los procesos del grupo del emisor
  - pid = -1: a todos los procesos cuyo UID real = UID efectivo del emisor. Si el emisor tiene un UID efectivo de superusuario se manda a todos excepto al proceso 0 y 1