TEMA 2. Procesos y Hebras

2.1 Generalidades sobre Procesos, Hilos y Planificación.

2.2. Diseño e implementación de procesos e hilos en Linux.

2.3 Planificación de la CPU en Linux.

Objetivos

- Conocer y diferenciar los conceptos de **proceso y hebra**
- Conocer los **estados** básicos de un proceso/hebra y las posibles transiciones entre los estados
- Saber en qué consiste un **cambio de contexto** y los costes que éste tiene para el sistema operativo
- Contenido y utilidad **de las estructuras de datos** que el SO utiliza para la gestión de los procesos/hebras
- Conocer y comparar las distintas implementaciones de las hebras
- Conocer la utilidad de la **planificación de procesos** y de los distintos planificadores que pueden existir
- Comparar los distintos algoritmos de planificación de CPU
- Conocer las operaciones básicas sobre procesos/hebras que pueden realizar los usuarios

Bibliografía

- [Sta05] W. Stallings, Sistemas Operativos. Aspectos Internos y Principios de Diseño (5/e), Prentice Hall, 2005.
- [Car07] J. Carretero et al., Sistemas Operativos (2ª Edición), McGraw-Hill, 2007.
- [Lov10] R. Love, *Linux Kernel Development (3/e)*, Addison-Wesley Professional, 2010.
- [Mau08] W. Mauerer, Professional Linux Kernel Architecture, Wiley, 2008.

Ejecución del SO

Núcleo fuera de todo proceso:

- Ejecuta el núcleo del sistema operativo fuera de cualquier proceso.
- El código del sistema operativo se ejecuta como una entidad separada que opera en modo privilegiado.

Ejecución dentro de los procesos de usuario:

- Software del sistema operativo en el contexto de un proceso de usuario.
- Un proceso se ejecuta en modo privilegiado cuando se ejecuta el código del sistema operativo.

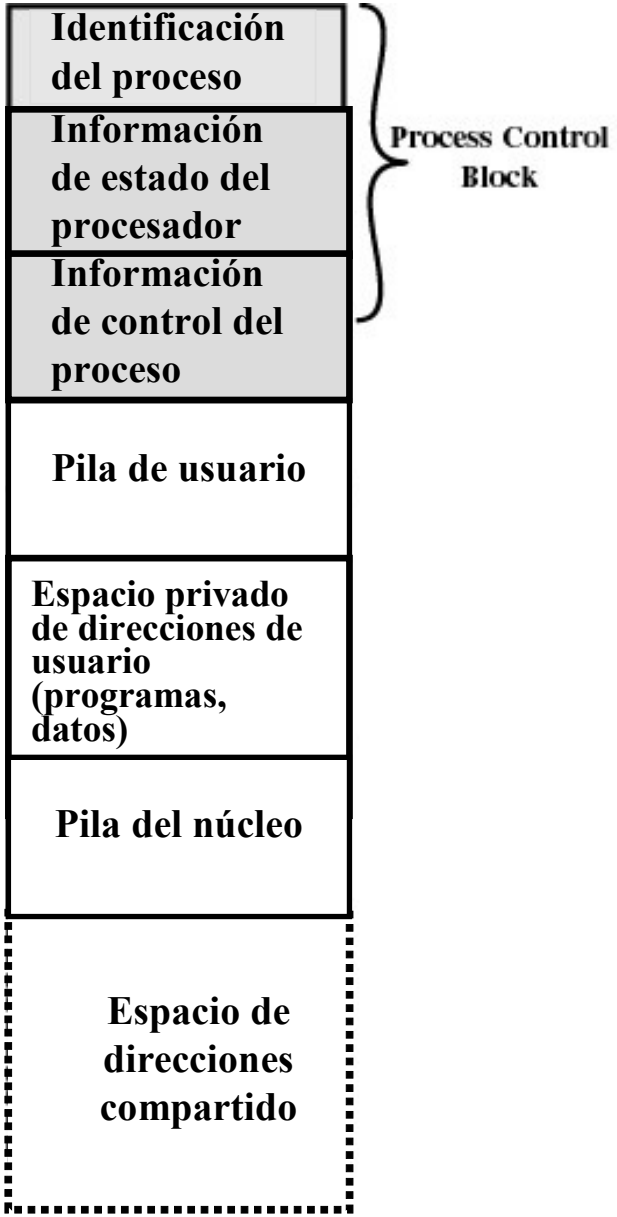


Figura 3.15. Imagen de un proceso: el sistema operativo se ejecuta dentro del proceso de usuario.

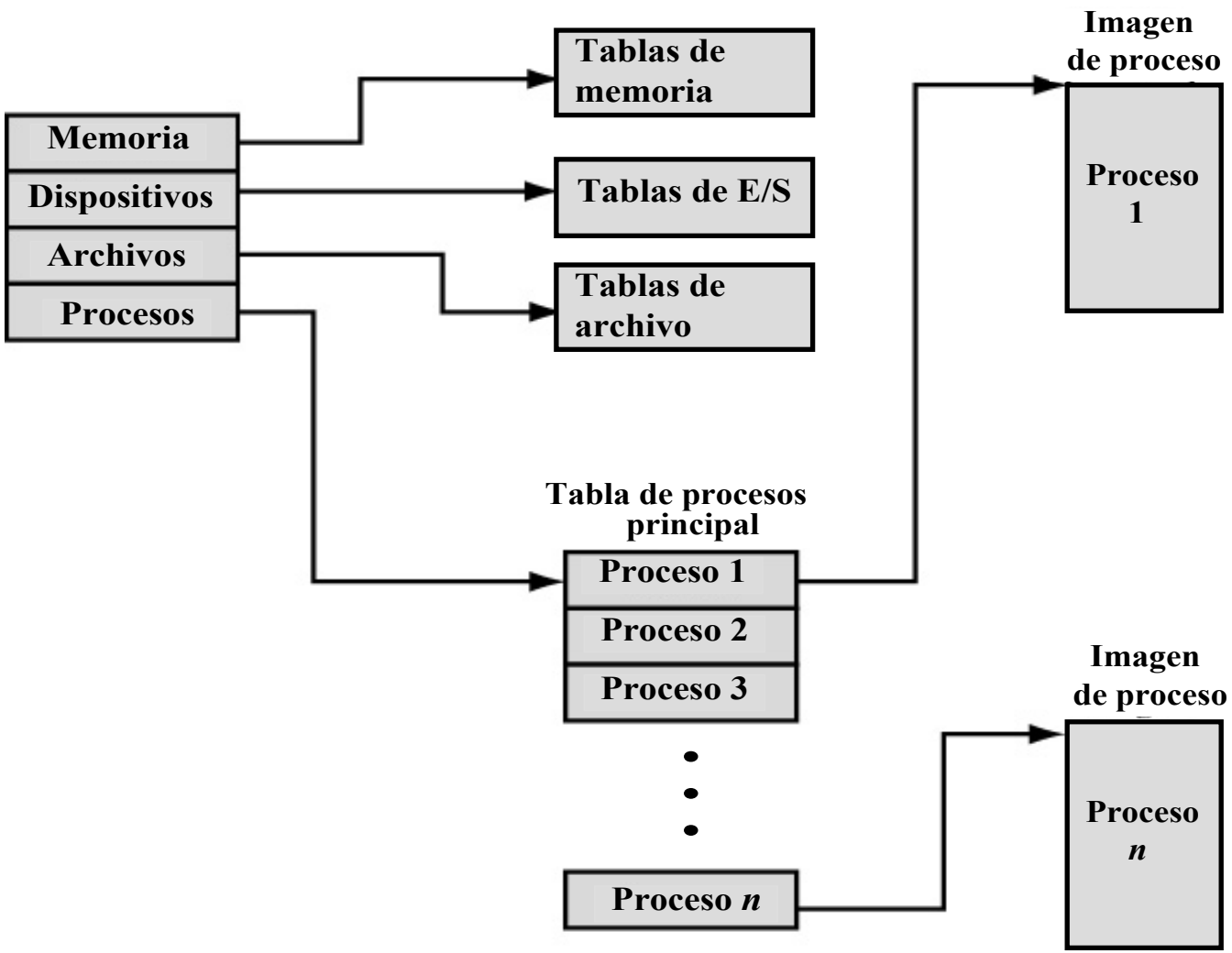


Figura 3.10. Estructura general de las tablas de control del sistema operativo.

Cambio de contexto

- Cuando un proceso esta ejecutándose, su PC, puntero a pila, registros, etc., están cargados en la CPU (es decir, los registros hardware contienen los valores actuales).
- Cuando el SO detiene un proceso ejecutándose, salva los valores actuales de estos registros (*contexto*) en el PCB de ese proceso.
- 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. Este trabajo es sobrecarga.

Cambio de contexto (y II)

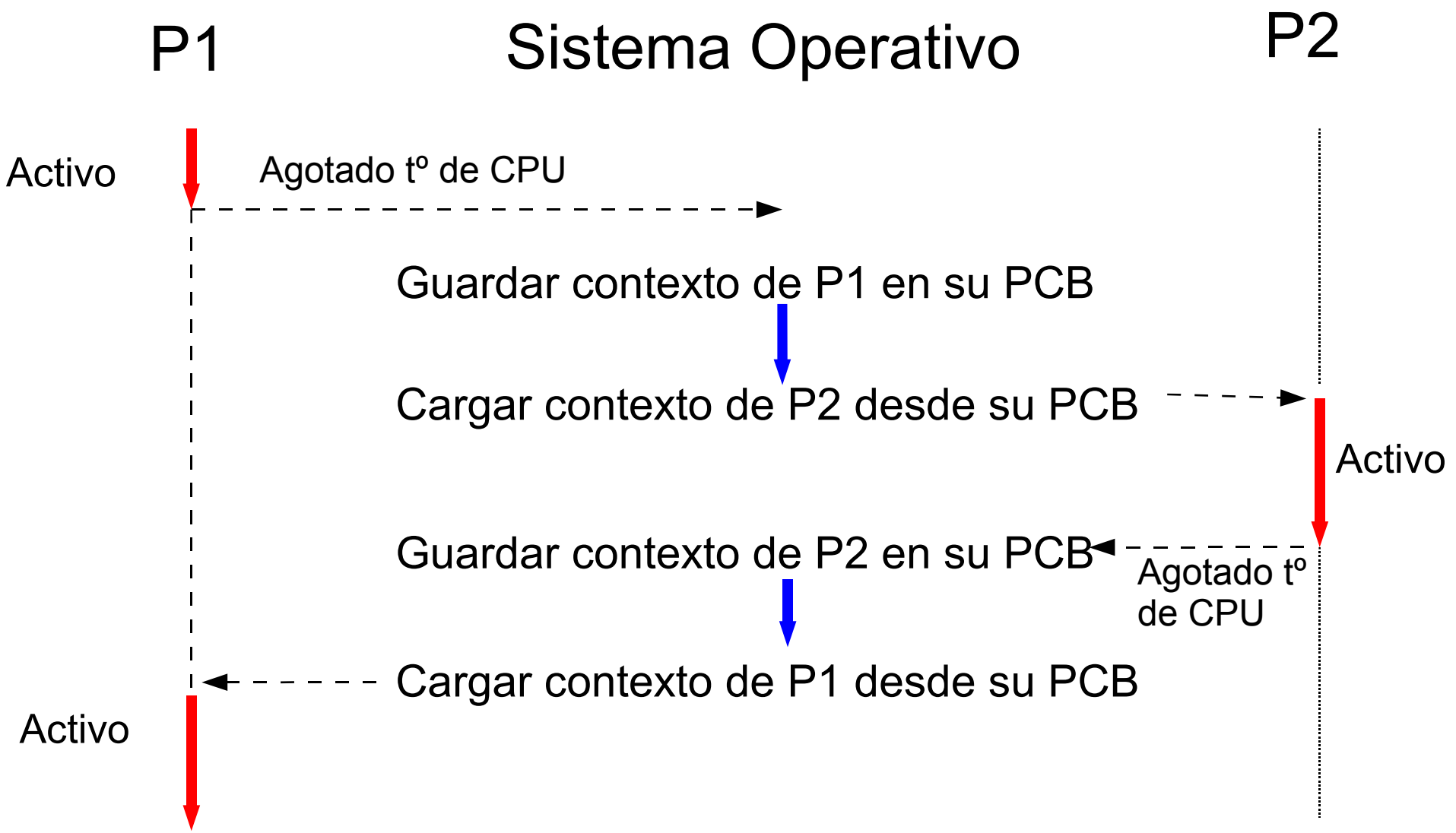
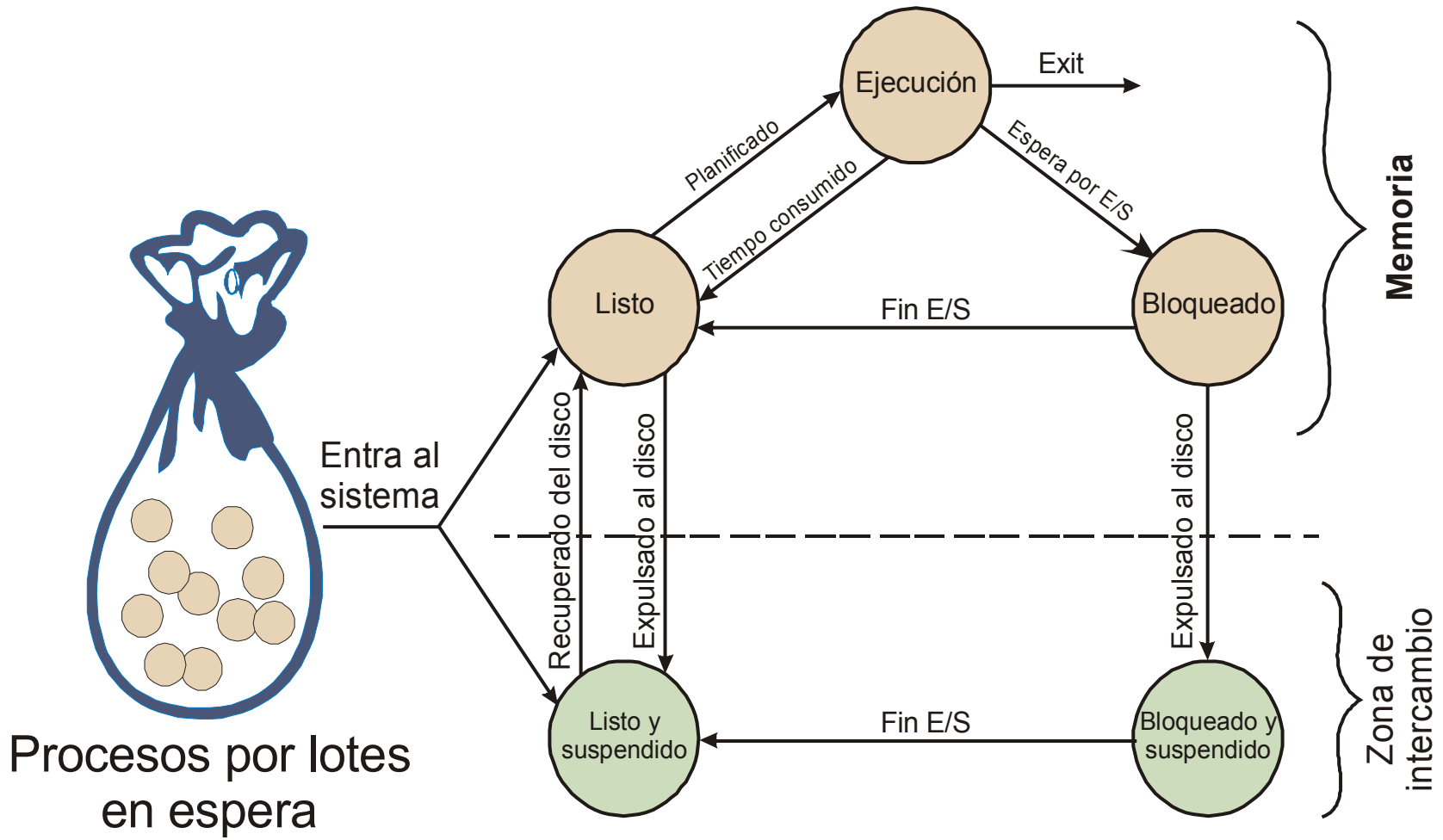


Diagrama de estados modificado



Operaciones sobre procesos
Creación de procesos

- ¿Qué significa crear un proceso?
  - » Asignarle el espacio de direcciones que utilizará
  - » Crear las estructuras de datos para su administración
- ¿Cuándo se crea? Los sucesos comunes son:
  - » En sistemas batch: en respuesta a la recepción y admisión de un trabajo
  - » En sistemas interactivos: cuando el usuario se conecta, el SO crea un proceso que ejecuta el intérprete de órdenes
  - » El SO puede crear un proceso para llevar a cabo un servicio solicitado por un proceso de usuario
  - » Un proceso puede crear otros procesos formando un árbol de procesos. Hablamos de relación padre-hijo (creador-creado)

Creación de procesos (y II)

- Cuando un proceso crea a otro, ¿cómo obtiene sus recursos el proceso hijo?
  - » Los obtiene directamente del SO: padre e hijo no comparten recursos.
  - » Comparte todos los recursos con el padre.
  - » Comparte un subconjunto de los recursos del padre.

Creación de procesos (y III)

- Ejecución:
  - » Padre e hijo se ejecutan concurrentemente.
  - » Padre espera al que el hijo termine.
- Espacio de direcciones:
  - » Hijo es un duplicado del padre (Unix, Linux).
  - » Hijo tiene un programa que lo carga (VMS, W2K)
- Ejemplo: En UNIX
  - » La llamada al sistema **`fork`** (bifurcar) crea un nuevo proceso.
  - » La llamada **`exec`** después de `fork` reemplaza el espacio de direcciones con el programa del nuevo proceso.

Creación de procesos (y IV)

Por tanto, ¿qué pasos, en general, deben realizarse en una operación de creación?

- Nombrar al proceso: asignarle un PID único
- Asignarle espacio (en MP y/o memoria secundaria)
- Crear el PCB e inicializarlo
- Insertarlo en la Tabla de procesos y ligarlo a la cola de planificación correspondiente
- Determinar su prioridad inicial

Terminación de procesos

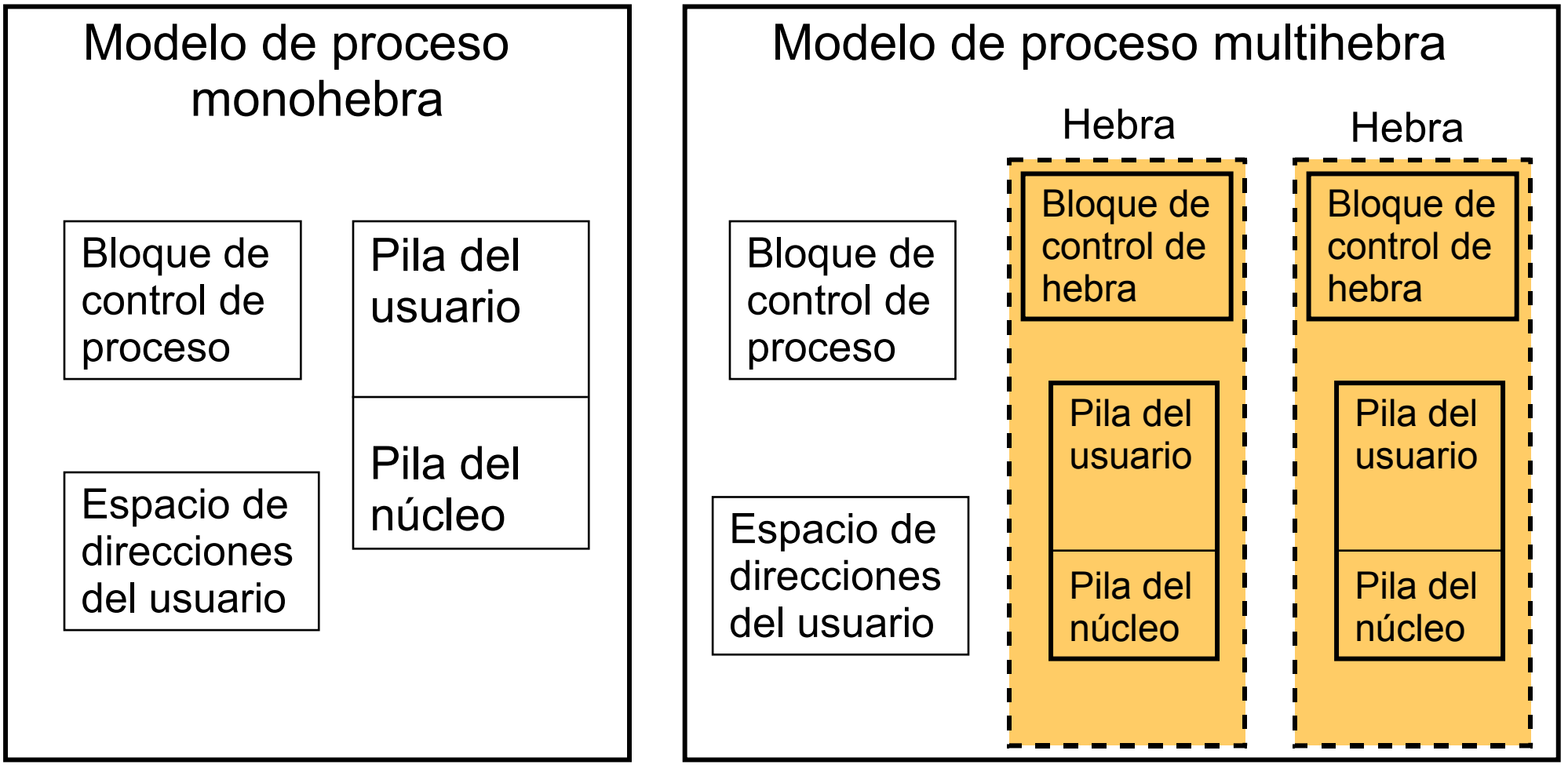
¿Qué sucesos determinan la finalización de un proceso?

- Cuando un proceso ejecuta la última instrucción, solicita al SO su finalización (**exit**)
  - Envío de datos del hijo al padre
  - Recursos del proceso son liberados por el SO
- El padre puede finalizar la ejecución de sus hijos (**abort o kill**)
  - El hijo ha sobrepasado los recursos asignados
  - La tarea asignada al hijo ya no es necesaria
  - El padre va a finalizar: el SO no permite al hijo continuar (terminación en cascada)
- El SO puede terminar la ejecución de un proceso porque se hayan producido errores o condiciones de fallo

Hebras
(*threads*, hilos o procesos ligeros)

- Una *hebra* (o *proceso ligero*) es la unidad básica de utilización de la CPU. Consta de:
  - » Contador de programa.
  - » Conjunto de registros.
  - » Espacio de pila.
  - » Estado
- Una hebra comparte con sus hebras pares una *tarea* que consiste en:
  - » Sección de código.
  - » Sección de datos.
  - » Recursos del SO (archivos abiertos, señales,..).
- Un *proceso pesado* o tradicional es igual a una tarea con una hebra.

Hebras (y II)


Modelo de proceso monohebra
Bloque de control de proceso
Pila del usuario
Pila del núcleo
Espacio de direcciones del usuario
Modelo de proceso multihebra
Hebra
Hebra
Bloque de control de proceso
Bloque de control de hebra
Bloque de control de hebra
Pila del usuario
Pila del usuario
Espacio de direcciones del usuario
Pila del núcleo
Pila del núcleo

Ventajas de las hebras

Se obtiene un mayor rendimiento y un mejor servicio debido a :

- Se reduce el tiempo de cambio de contexto, el tiempo de creación y el tiempo de terminación.
- En una tarea con múltiples hebras, mientras una hebra esta bloqueada y esperando, una segunda hebra de la misma tarea puede ejecutarse (depende del tipo de hebras).
- La comunicación entre hebras de una misma tarea se realiza a través de la memoria compartida (no necesitan utilizar los mecanismos del núcleo).
- Las aplicaciones que necesitan compartir memoria se benefician de la hebras.

Funcionalidad de las hebras

Al igual que los procesos la hebras poseen un estado de ejecución y pueden sincronizarse.

- **Estados de las hebras**: Ejecución, Lista o Preparada y Bloqueada.
- Operaciones básicas relacionadas con el **cambio de estado** en hebras:
  - Creación
  - Bloqueo
  - Desbloqueo
  - Terminación
- **Sincronización entre hebras**.

Tipos de hebras

Tipos de hebras: *Núcleo (Kernel) , de usuario y enfoques híbridos*

Hebras de usuario

- Todo el trabajo de gestión de hebras lo realiza la aplicación, el núcleo no es consciente de la existencia de hebras.
- Se implementan a través de una biblioteca en el nivel usuario. La biblioteca contiene código para gestionar las hebras:
  - crear hebras, intercambiar datos entre hebras, planificar la ejecución de las hebras y salvar y restaurar el contexto de las hebras.
- La unidad de planificación para el núcleo es el proceso

Tipos de hebras (y II)

Hebras Kernel (Núcleo)

- Toda la gestión de hebras lo realiza el núcleo.
- El SO proporciona un conjunto de llamadas al sistema similares a las existentes para los procesos (Mach, OS/2).
- El núcleo mantiene la información de contexto del proceso como un todo y de cada hebra.
- La unidad de planificación es la hebra.
- Las propias funciones del núcleo pueden ser multihebras.

Tipos de hebras (y III)

- Ventajas del uso de las hebras tipo usuario frente a las de tipo núcleo:
  - » Se evita la sobre carga de cambios de modo, que sucede cada vez que se pasa el control de una hebra a otra en sistemas que utilizan hebras núcleo.
  - » Se puede tener una planificación para las hebras distinta a la planificación subyacente del SO.
  - » Se pueden ejecutar en cualquier SO. Su utilización no supone cambio en el núcleo.

Tipos de hebras (y IV)

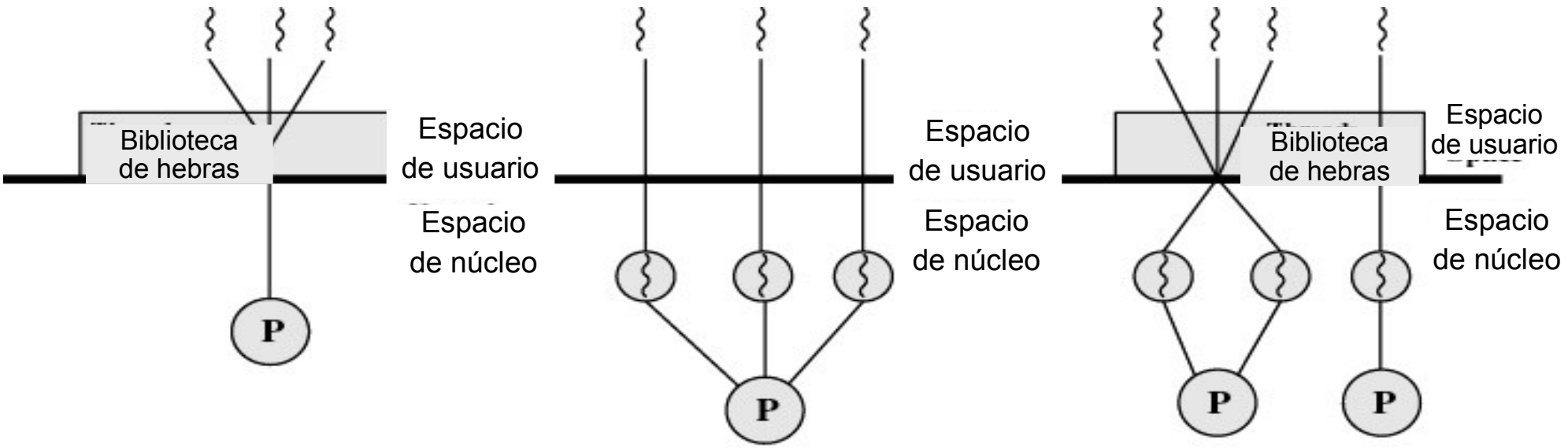
- Desventajas del uso de las hebras tipo usuario frente a las de tipo núcleo.
  - » Cuando un proceso realiza una llamada al sistema bloqueadora no sólo se bloquea la hebra que realizó la llamada, sino todas las hebras del proceso.
  - » En un entorno multiprocesador, una aplicación mutihebra no puede aprovechar la ventajas de dicho entorno ya que el núcleo asigna un procesador a un proceso.

Tipos de hebras (y V)

Enfoques híbridos

- Implementan tanto hebras a nivel *kernel* como usuario (p. ej. Solaris 2).
- La creación de hebras, y la mayor parte de la planificación y sincronización se realizan en el espacio de usuario.
- Las distintas hebras de una aplicación se asocian con varias hebras del núcleo (mismo o menor número), el programador puede ajustar la asociación para obtener un mejor resultado.
- Las múltiples hebras de una aplicación se pueden paralelizar y las llamadas al sistema bloqueadoras no necesitan bloquear todo el proceso.

Tipos de hebras (y VI)



(a) Nivel de usuario puro **(b) Nivel de núcleo puro** **(c) Sistema Híbrido**

Hebra a nivel de usuario Hebra a nivel de núcleo P Proceso

Hebras a nivel de usuario y a nivel de núcleo (*Stalling*)

Planificación: PCB's y Colas de Estados

- El SO mantiene una colección de colas que representan el estado de todos los procesos en el sistema.
- Típicamente hay una cola por estado.
- Cada PCB esta encolado en una 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.

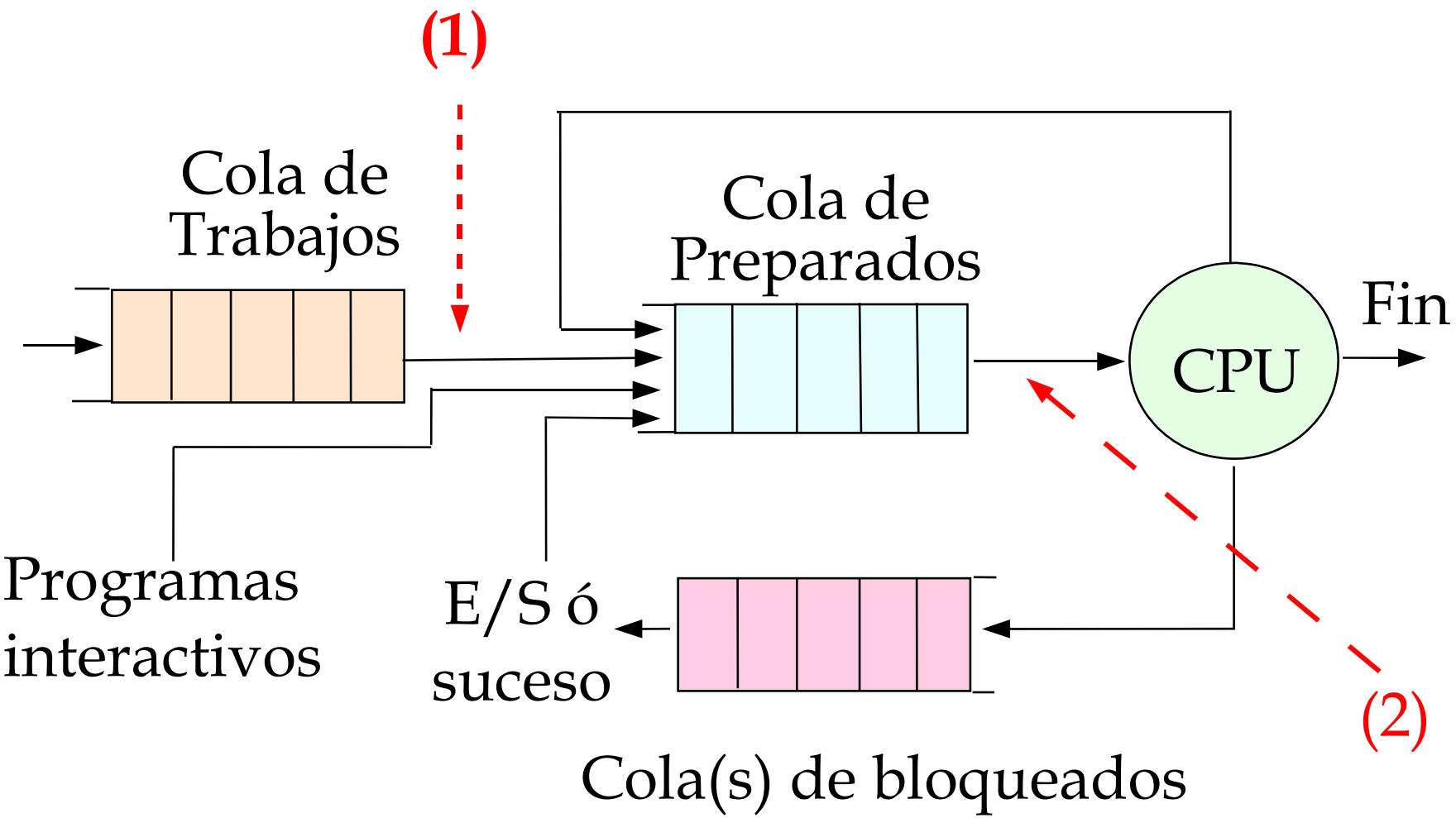
Colas de estados

- *Cola de trabajos* -- conjunto de los trabajos pendientes de ser admitidos en el sistema (trabajos por lotes que no están en memoria).
- *Cola de preparados* -- conjunto de todos los procesos que residen en memoria principal, preparados y esperando para ejecutarse.
- *Cola(s) de bloqueados* -- conjunto de todos los procesos esperando por un dispositivo de E/S particular o por un suceso.

Tipos de planificadores

- Planificador Parte del SO que controla la utilización de un recurso.
- Tipos de planificadores de la CPU:
  - » *Planificador a largo plazo* (planificador de trabajos): selecciona los procesos que deben llevarse a la cola de preparados; (1) en figura siguiente.
  - » *Planificador a corto plazo* (planificador de la CPU): selecciona al proceso que debe ejecutarse a continuación, y le asigna la CPU; (2) en figura siguiente.
  - » *Planificador a medio plazo*.

Migración entre colas



Características de los planificadores

- *El planificador a corto plazo* :

»trabaja con la cola de preparados.

»se invoca muy frecuentemente (milisegundos) por lo que debe ser rápido.

- *El planificador a largo plazo*

»Permite controlar el *grado de multiprogramación*.

»se invoca poco frecuentemente (segundos o minutos), por lo que puede ser más lento.

Clasificación de procesos

- *Procesos limitados por E/S o procesos cortos* -- dedican más tiempo a realizar E/S que computo; muchas ráfagas de CPU cortas y largos períodos de espera.

- *Procesos limitados por CPU o procesos largos* -- dedican más tiempo en computación que en realizar E/S; pocas ráfagas de CPU pero largas.

Mezcla de trabajos

Es importante que el planificador a largo plazo seleccione una buena mezcla de trabajos, ya que:

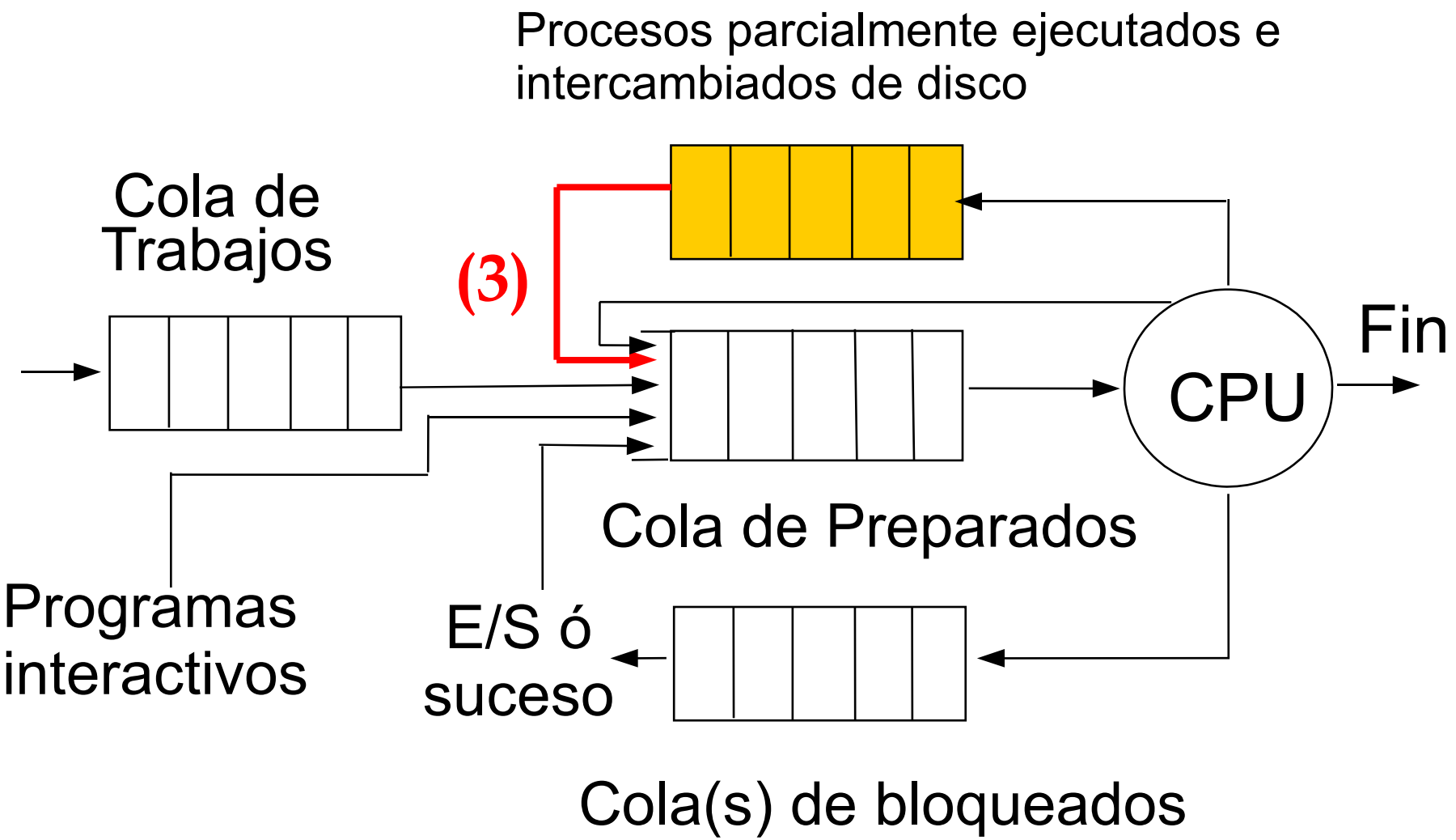
- Si todos los trabajos están limitados por E/S, la cola de preparados estará casi siempre vacía y el planificador a corto plazo tendrá poco que hacer.

- Si todos los procesos están limitados por CPU, la cola de E/S estará casi siempre vacía y el sistema estará de nuevo desequilibrado.

Planificador a medio plazo

- En algunos SO's, p. ej. SO de tiempo compartido, a veces es necesario sacar procesos de la memoria (reducir *el grado de multiprogramación*), bien para mejorar la mezcla de procesos, bien por cambio en los requisitos de memoria, y luego volverlos a introducir (intercambio o *swapping*).
- El *planificador a medio plazo* se encarga de devolver los procesos a memoria. Transición (3) en la siguiente figura.

Planificador a medio plazo (y II)



Despachador

- El despachador (*dispatcher*) es el módulo del SO que da el control de la CPU al proceso seleccionado por el planificador a corto plazo; esto involucra:
  - » Cambio de contexto (se realiza en modo kernel).
  - » Conmutación a modo usuario.
  - » Salto a la posición de memoria adecuada del programa para su reanudación.
- *Latencia de despacho* -- tiempo que emplea el despachador en detener un proceso y comenzar a ejecutar otro.

Activación del Despachador

El despachador puede actuar cuando:

1. Un proceso no quiere seguir ejecutándose (finaliza o ejecuta una operación que lo bloquea)
2. Un elemento del SO determina que el proceso no puede seguir ejecutándose (ej. E/S o retiro de memoria principal)
3. El proceso agota el quantum de tiempo asignado
4. Un suceso cambia el estado de un proceso de bloqueado a ejecutable

Políticas de planificación: Monoprocesadores

- Objetivos:
  - » Buen rendimiento (productividad)
  - » Buen servicio
- Para saber si un proceso obtiene un buen servicio, definiremos un conjunto de medidas: dado un proceso *P*, que necesita un tiempo de servicio (ráfaga) *t*
  - *Tiempo de respuesta (T)*-- tiempo transcurrido desde que se remite una solicitud (entra en la cola de preparados) hasta que se produce la primera respuesta (no se considera el tiempo que tarda en dar la salida)
  - *Tiempo de espera (M)*-- tiempo que un proceso ha estado esperando en la cola de preparados: T - t
  - *Penalización (P)* -- T / t
  - *Indice de respuesta (R)* -- t / T : fracción de tiempo que P está recibiendo servicio.

Políticas de planificación: Monoprocesadores (y II)

Otras medidas interesantes son:

- *Tiempo del núcleo* -- tiempo perdido por el SO tomando decisiones que afectan a la planificación de procesos y haciendo los cambios de contexto necesarios. En un sistema eficiente debe representar entre el 10% y el 30% del total del tiempo del procesador
- *Tiempo de inactividad* -- tiempo en el que la cola de ejecutables está vacía y no se realiza ningún trabajo productivo
- *Tiempo de retorno* – cantidad de tiempo necesario para ejecutar un proceso completo

Políticas de planificación: Monoprocesadores (III)

- Las políticas de planificación se comportan de distinta manera dependiendo de la clase de procesos
  - » Ninguna política de planificación es completamente satisfactoria, cualquier mejora en una clase de procesos es a expensas de perder eficiencia en los procesos de otra clase.
- Podemos clasificarlas en:
  - » **No apropiativas (no expulsivas)**: una vez que se le asigna el procesador a un proceso, no se le puede retirar hasta que éste voluntariamente lo deje (finalice o se bloquee)
  - » **Apropiativas (expulsivas)**: al contrario, el SO puede apropiarse del procesador cuando lo decida

Políticas de planificación de la CPU (y IV)

- FCFS
- El más corto primero:
  - no apropiativo
  - apropiativo
- Planificación por prioridades
- Por turnos (*Round-Robin*)
- Colas múltiples
- Colas múltiples con realimentación.

FCFS (*First Come First Served)*

- Los procesos son servidos según el orden de llegada a la cola de ejecutables.
- Es *no apropiativo*, cada proceso se ejecutará hasta que finalice o se bloquee.
- Fácil de implementar pero pobre en cuanto a prestaciones.
- Todos los procesos pierden la misma cantidad de tiempo esperando en la cola de ejecutables independientemente de sus necesidades.
- Procesos cortos muy penalizados.
- Procesos largos poco penalizados.

El más corto primero (SJF)

- Es *no apropiativo.*
- Cuando el procesador queda libre, selecciona el proceso que requiera un tiempo de servicio menor.
- Si existen dos o más procesos en igualdad de condiciones, se sigue FCFS.
- Necesita conocer explícitamente el tiempo estimado de ejecución (tº servicio) ¿Cómo?.
- Disminuye el tiempo de respuesta para los procesos cortos y discrimina a los largos.
- Tiempo medio de espera bajo.

El más corto primero apropiativo (SRTF)

- Cada vez que entra un proceso a la cola de ejecutables se comprueba si su tiempo de servicio es menor que el tiempo de servicio que le queda al proceso que está ejecutándose. Casos:
  - » **Si es menor**: se realiza un cambio de contexto y el proceso con menor tiempo de servicio es el que se ejecuta.
  - » **No es menor**: continúa el proceso que estaba ejecutándose.
- El tiempo de respuesta es menor excepto para procesos muy largos.
- Se obtiene la menor penalización en promedio (mantiene la cola de ejecutables con la mínima

Planificación por prioridades

- Asociamos a cada proceso un número de prioridad (entero).
- Se asigna la CPU al proceso con mayor prioridad (enteros menores = mayor prioridad)
  - Apropiativa
  - No apropiativa
- **Problema**: Inanición -- los procesos de baja prioridad pueden no ejecutarse nunca.
- **Solución**: Envejecimiento -- con el paso del tiempo se incrementa la prioridad de los procesos.

Por Turnos (Round-Robin)

- La CPU se asigna a los procesos en intervalos de tiempo (quatum).
- Procedimiento:
  - » Si el proceso finaliza o se bloquea antes de agotar el quantum, libera la CPU. Se toma el siguiente proceso de la cola de ejecutables (la cola es FIFO) y se le asigna un quantum completo.
  - » Si el proceso no termina durante ese quantum, se interrumpe y se coloca al final de la cola de ejecutables.
- Es apropiativo.

***Nota**: En los ejemplos* supondremos que si un proceso *A* llega a la cola de ejecutables al mismo tiempo que otro *B* agota su quantum, la llegada de *A* a la cola de ejecutables ocurre antes de que *B* se incorpore a ella.

Por Turnos (y III)

- Los valores típicos del quantum están entre 1/60sg y 1sg.
- Penaliza a todos los procesos en la misma cantidad, sin importar si son cortos o largos.
- Las ráfagas muy cortas están más penalizadas de lo deseable.
- ¿valor del quantum?

  - muy grande (excede del tº de servicio de todos los procesos) □ se convierte en FCFS

  - muy pequeño □ el sistema monopoliza la CPU haciendo cambios de contexto (tº del núcleo muy alto)

Colas múltiples

- La cola de preparados se divide en varias colas y cada proceso es asignado permanentemente a una cola concreta P. ej. interactivos y batch
- Cada cola puede tener su propio algoritmo de planificación P. ej. interactivos con RR y batch con FCFS
- Requiere una planificación entre colas
  - » Planificación con prioridades fijas. P. ej. primero servimos a los interactivos luego a los batch
  - » Tiempo compartido -- cada cola obtiene cierto tiempo de CPU que debe repartir entre sus procesos. P. ej. 80% interactivos en RR y 20% a los batch con FCFS

Colas múltiples con realimentación

- Un proceso se puede mover entre varias colas
- Requiere definir los siguientes parámetros:
  - » Número de colas
  - » Algoritmo de planificación para cada cola
  - » Método utilizado para determinar cuando trasladar a un proceso a otra cola
  - » Método utilizado para determinar en qué cola se introducirá un proceso cuando necesite un servicio
  - » Algoritmo de planificación entre colas
- Mide en tiempo de ejecución el comportamiento real de los procesos
- Disciplina de planificación más general (Unix, Linux Windows NT)

Planificación en multiprocesadores

- Tres aspectos interrelacionados:
  - Asignación de procesos a procesadores
    - Cola dedicada para cada procesador
    - Cola global para todos los procesadores
  - Uso de multiprogramación en cada procesador individual
  - Activación del proceso

Planificación en multiprocesadores (y II)

- Planificación de procesos

  → igual que en monoprocesadores pero teniendo en cuenta:

  - Número de CPUs
  - Asignación/Liberación proceso-procesador

- Planificación de hilos

  → permiten explotar el paralelismo real dentro de una aplicación

Planificación de hilos en multiprocesadores

1) Compartición de carga

- Cola global de hilos preparados
- Cuando un procesador está ocioso, se selecciona un hilo de la cola (método muy usado)

2) Planificación en pandilla

- Se planifica un conjunto de hilos afines (de un mismo proceso) para ejecutarse sobre un conjunto de procesadores al mismo tiempo (relación 1 a 1)
- Útil para aplicaciones cuyo rendimiento se degrada mucho cuando alguna parte no puede ejecutarse (los hilos necesitan sincronizarse)

Planificación de hilos en multiprocesadores

3) Asignación de procesador dedicado

- Cuando se planifica una aplicación, se asigna un procesador a cada uno de sus hilos hasta que termine la aplicación
- Algunos procesadores pueden estar ociosos → No hay multiprogramación de procesadores

4) Planificación dinámica

- La aplicación permite que varíe dinámicamente el número de hilos de un proceso
- El SO ajusta la carga para mejorar la utilización de los procesadores

Sistemas de tiempo real

- La exactitud del sistema no depende sólo del resultado lógico de un cálculo sino también del instante en que se produzca el resultado.
- Las tareas o procesos intentan controlar o reaccionar ante sucesos que se producen en “tiempo real” (eventos) y que tienen lugar en el mundo exterior.
- Características de las tareas de tiempo real:
  - Tarea de tº real duro: debe cumplir su plazo límite
  - Tarea de tº real suave: tiene un tiempo límite pero no es obligatorio
  - Periódicas: se sabe cada cuánto tiempo se tiene que ejecutar
  - Aperiódicas: tiene un plazo en el que debe comenzar o acabar o restricciones respecto a esos tiempos pero son impredecibles

Planificación de sistemas de tº real

- Los distintos enfoques dependen de:
  - Cuándo el sistema realiza un análisis de viabilidad de la planificación
    - Estudia si puede atender a todos los eventos periódicos dado el tiempo necesario para ejecutar la tarea y el periodo
  - Si se realiza estática o dinámicamente
  - Si el resultado del análisis produce un plan de planificación o no

Planificación en sistemas de tº real (y II)

- Enfoques estáticos dirigidos por tabla
  - Análisis estático que genera una planificación que determina cuándo empezará cada tarea
- Enfoques estáticos expulsivos dirigidos por prioridad
  - Análisis estático que no genera una planificación, sólo se usa para dar prioridad a las tareas. Usa planificación por prioridades
- Enfoques dinámicos basados en un plan
  - Se determina la viabilidad en tº de ejecución (dinámicamente): se acepta una nueva tarea si es posible satisfacer sus restricciones de tº
- Enfoques dinámicos de menor esfuerzo (el más usado)
  - No se hace análisis de viabilidad. El sistema intenta cumplir todos los plazos y aborta ejecuciones si su plazo ha finalizado

Problema: Inversión de Prioridad

- Se da en un esquema de planificación de prioridad y cuando:
  - Una tarea de mayor prioridad espera por una tarea (proceso) de menor prioridad debido al bloqueo de un recurso de uso exclusivo (no compartible)
- Enfoques para evitarla:
  - Herencia de prioridad: la tarea menos prioritaria hereda la prioridad de la tarea más prioritaria
  - Techo de prioridad: Se asocia una prioridad a cada recurso de uso exclusivo que es más alta que cualquier prioridad que pueda tener una tarea, y esa prioridad se le asigna a la tarea a la que se le da el recurso
  - En ambos, la tarea menos prioritaria vuelve a tener el valor de prioridad que tenía cuando libere el recurso

Diseño e implementación de procesos e hilos en Linux

Nos basamos el kernel 2.6 de Linux.
Podemos descargar los fuentes de www.kernel.org

- El núcleo identifica a los procesos (tareas - tasks) por su **PID**
- En Linux, proceso es la entidad que se crea con la llamada al sistema ***fork*** (excepto el proceso 0) y ***clone***
- Procesos especiales que existen durante la vida del sistema:
  - **Proceso 0**: creado "a mano" cuando arranca el sistema. Crea al proceso 1.
  - **Proceso 1 (Init)**: antecesor de cualquier proceso del sistema

Linux: estructura task

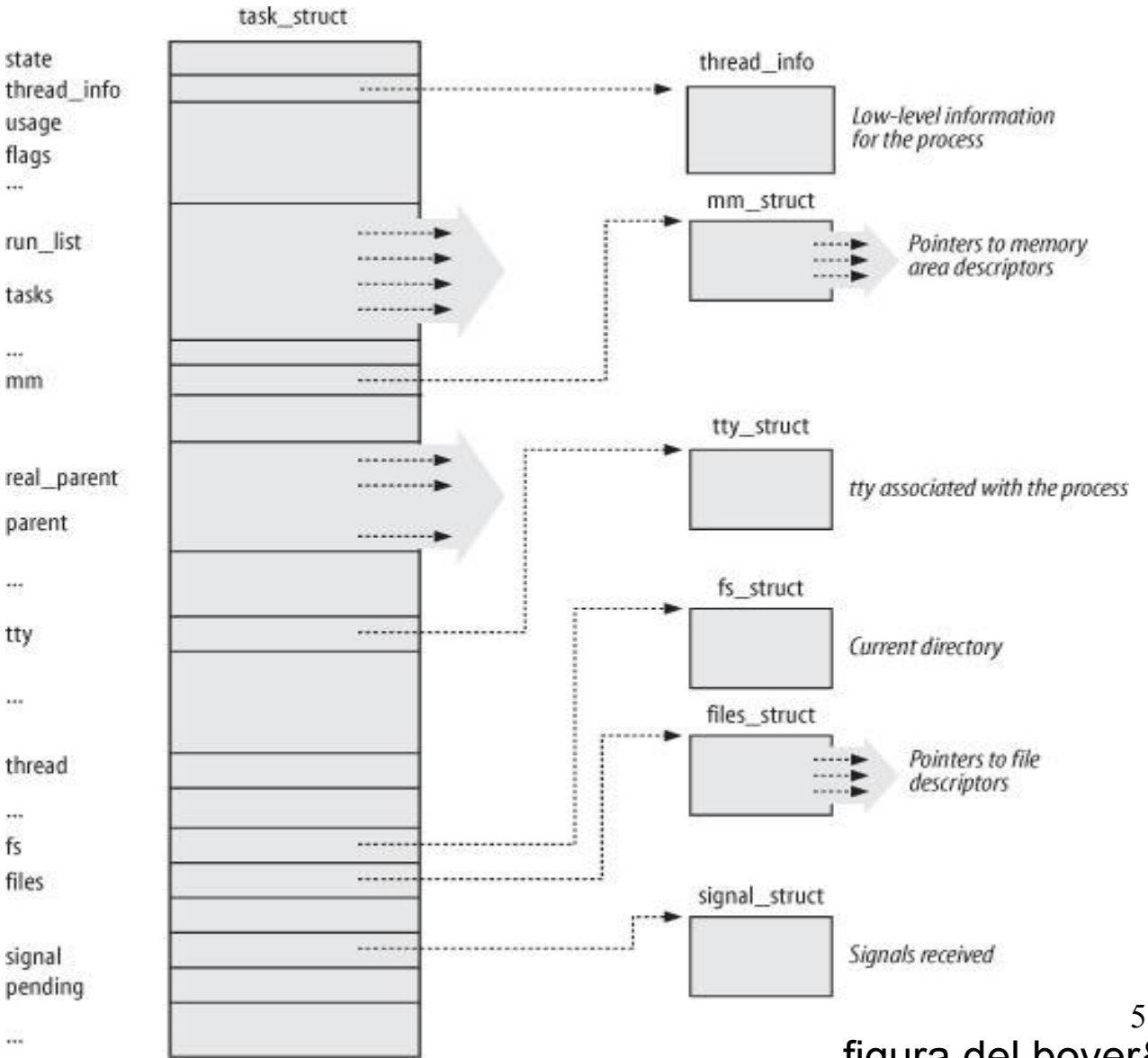
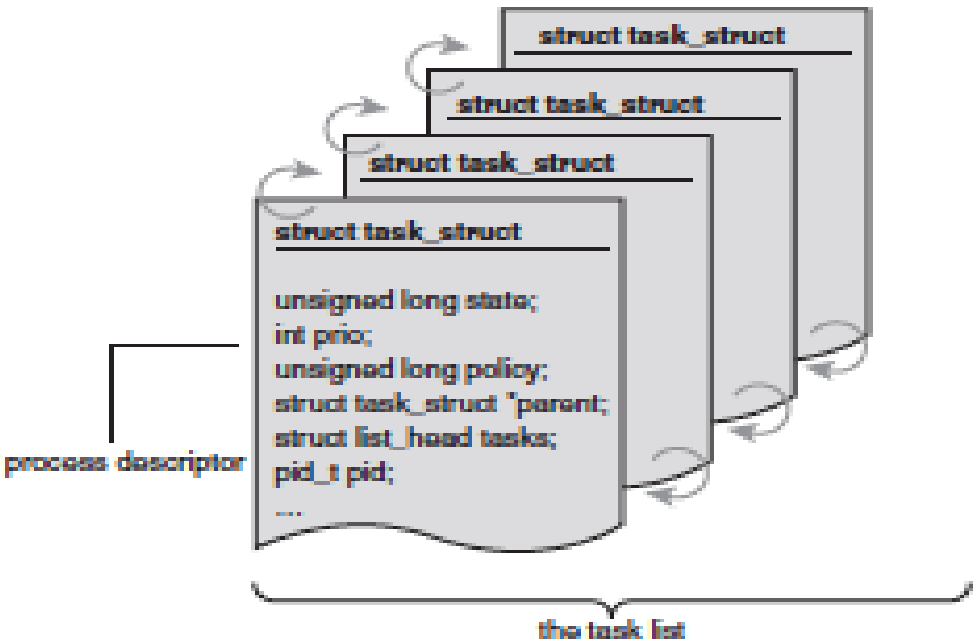


figura del bover&cesati

Linux: estructura task (y II)

- El Kernel almacena la lista de procesos como una lista circular doblemente enlazada: `task list`
- Cada elemento es un descriptor de proceso (PCB) definido en `</include/linux/sched.h>`



Linux: estructura task (y III)

```
struct task_struct {  /// del kernel 2.6.24
    volatile long state;  /* -1 unrunnable, 0 runnable, >0 stopped */
    /*...*/
/* Informacion para planificacion */
    int prio, static_prio, normal_prio;
    struct list_head run_list;
    const struct sched_class *sched_class;
    struct sched_entity se;
/*...*/
    unsigned int policy;
    cpumask_t cpus_allowed;
    unsigned int time_slice;
/*...*/
```

Linux: estructura task (y IV)

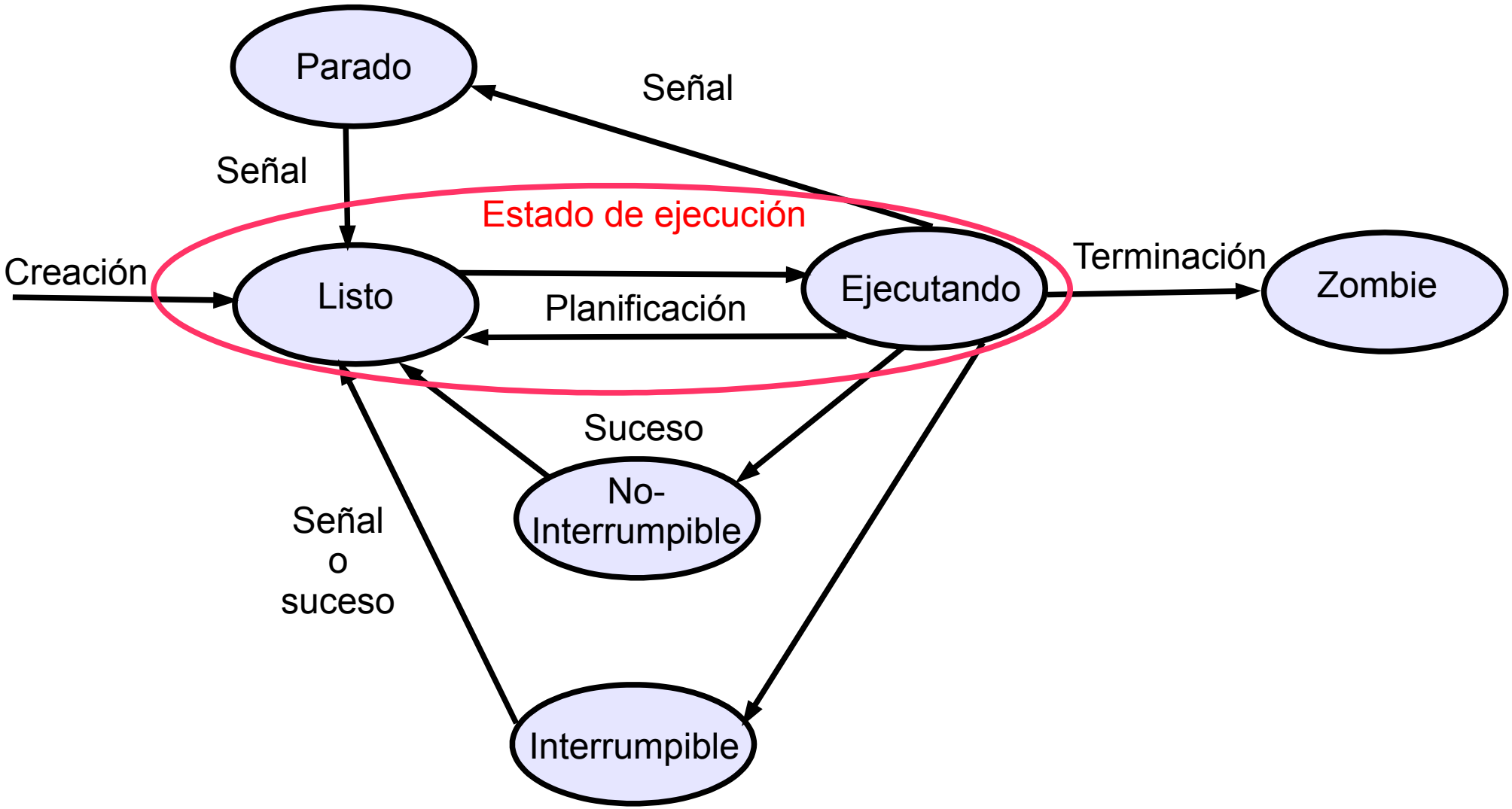
```
//* Memoria asociada a la tarea */
     struct mm_struct *mm, *active_mm;
/*...*/
     pid_t pid;
/* Relaciones entre task_struct */
     struct task_struct *parent; /* parent process */
     struct list_head children; /* list of my children */
     struct list_head sibling; /* linkage in my parent's children list */
/* Informacion para planificacion y señales */
     unsigned int rt_priority;
     sigset_t blocked, real_blocked;
     sigset_t saved_sigmask; /* To be restored with TIF_RESTORE_SIGMASK */
     struct sigpending pending;
/*...*/
}
```

Estados de un proceso en Linux

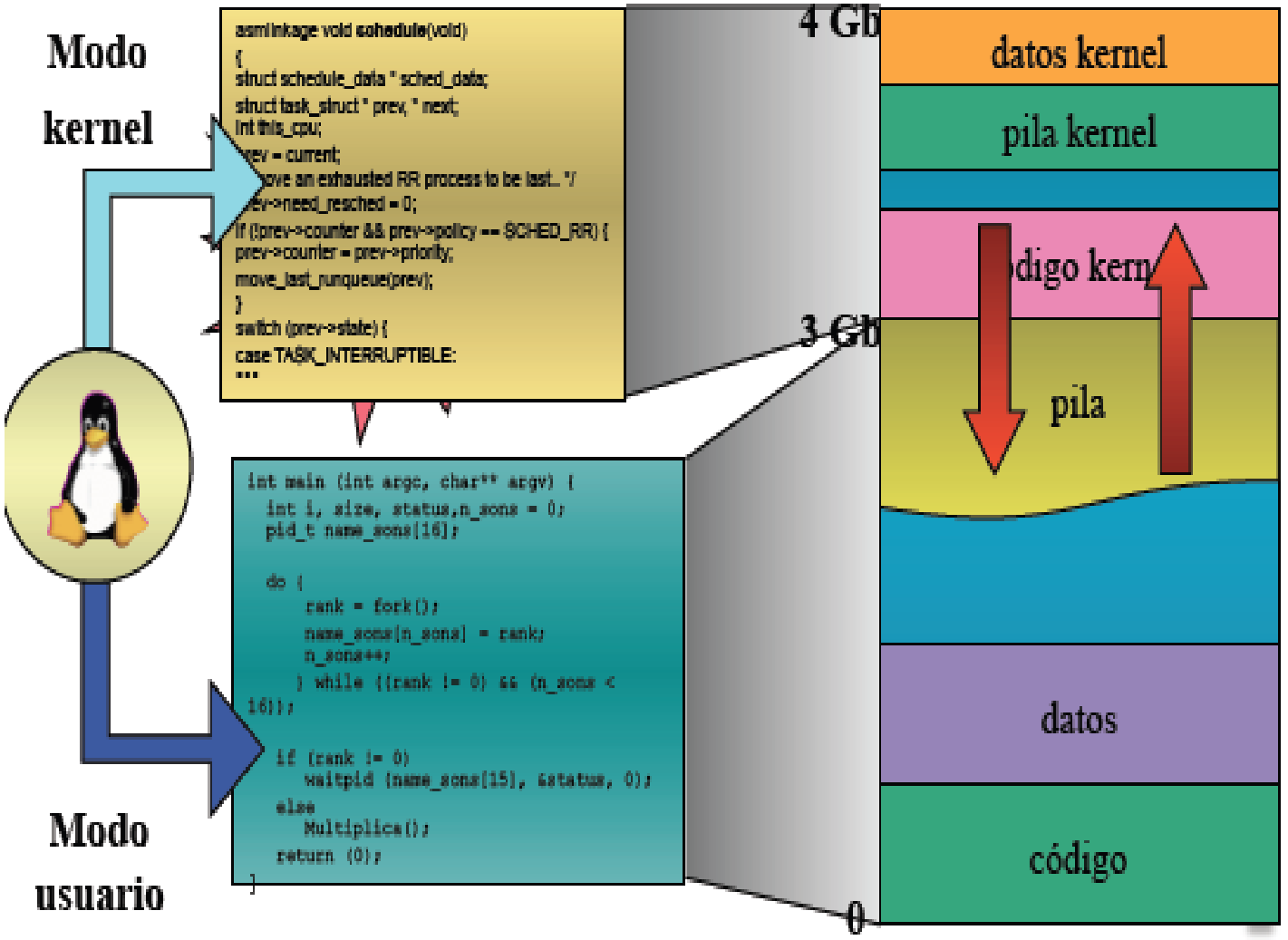
La variable state de task_struct especifica el estado actual de un proceso.

Ejecución TASK_RUNNING	Se corresponde con dos: ejecutándose o preparado para ejecutarse (en la cola de procesos preparados)
Interrumpible TASK_INTERRUPTIBLE	El proceso está bloqueado y sale de este estado cuando ocurre el suceso por el cual está bloqueado o porque le llegue una señal
No interrumpible TASK_UNINTERRUPTIBLE	El proceso está bloqueado y sólo cambiará de estado cuando ocurra el suceso que está esperando (no acepta señales)
Parado TASK_STOPPED	El proceso ha sido detenido y sólo puede reanudarse por la acción de otro proceso (ejemplo, proceso parado mientras está siendo depurado)
TASK_TRACED	El proceso está siendo traceado por otro proceso
Zombie EXIT_ZOMBIE	El proceso ya no existe pero mantiene la entrada de la tabla de procesos hasta que el padre haga un wait (**EXIT_DEAD**)

Modelo de procesos/hilos en Linux



Espacio de direcciones de un proceso en Linux



EL árbol de procesos (y II)

Cada `task_struct` tiene un puntero ...

a la task_struct de su padre:

```
struct task_struct *parent
```

a una lista de hijos (llamada children):

```
struct list_head children;
```

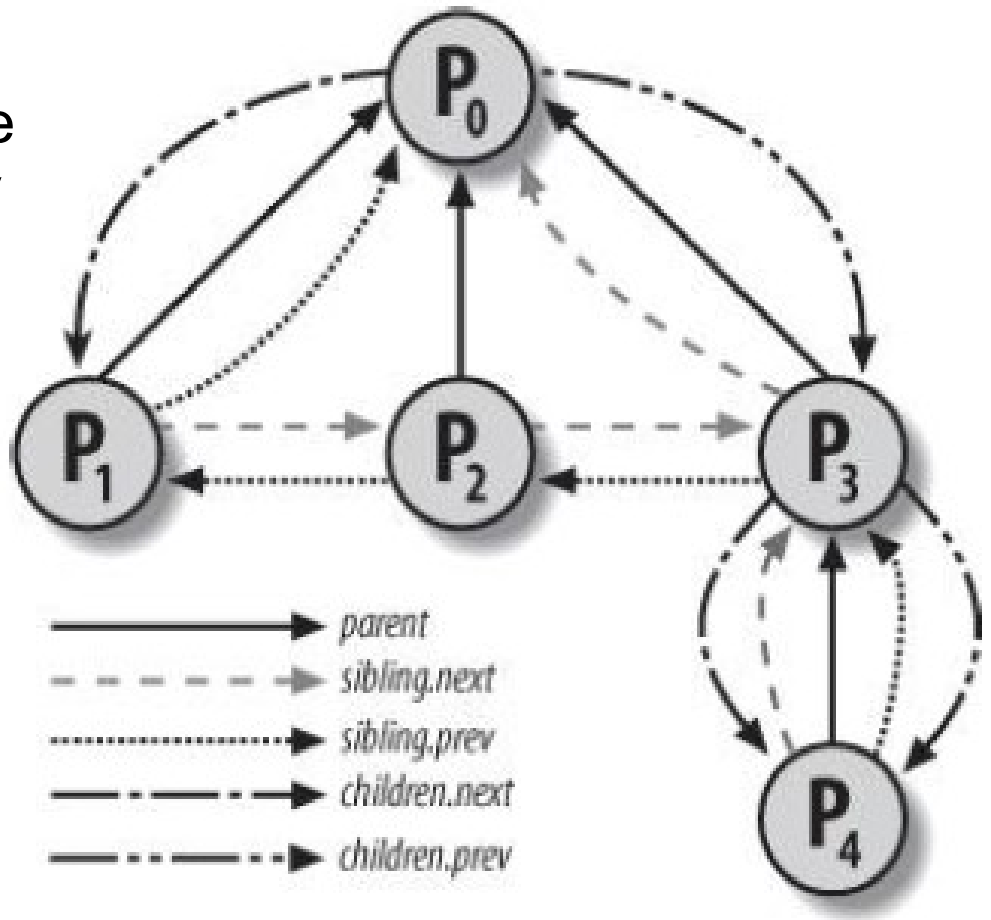
y a una lista de sus hermanos (llamada sibling):

```
struct list_head sibling
```

El kernel dispone de procedimientos eficaces para las acciones usuales de manipulación de la lista. (Para una explicación detallada de `list_head` ver [Mau08 1.3.13])

El árbol de procesos

- Si el proceso P0 hace una llamada la sistema fork, genera el proceso P1, se dice que P0 es el proceso padre y P1 es el hijo.
- Si el proceso P0 hace varios fork general de varios proceso hijos P1,P2,P3, la relación entre ellos es de hermanos (sibling)
- Todos los procesos son descendientes del proceso init (cuyo PID es 1)



Implementación de hilos en Linux

- Desde el punto de vista del kernel no hay distinción entre hebra y proceso
- Linux implementa el concepto de hebra como un proceso sin más, que simplemente comparte recursos con otros procesos
- Cada hebra tiene su propia **`task_struct`**
- La llamada al sistema **`clone`** crea un nuevo proceso o hebra

```
#include <sched.h>

int clone (int (*fn) (void *), void *child_stack, int flags,
           void *arg);
```

Llamada `clone`

Flag	Meaning
CLONE_FILES	Parent and child share open files.
CLONE_FS	Parent and child share filesystem information.
CLONE_IDLETASK	Set PID to zero (used only by the idle tasks).
CLONE_NEWNS	Create a new namespace for the child.
CLONE_PARENT	Child is to have same parent as its parent.
CLONE_PTRACE	Continue tracing child.
CLONE_SETTID	Write the TID back to user-space.
CLONE_SETTLS	Create a new TLS for the child.
CLONE_SIGHAND	Parent and child share signal handlers and blocked signals.
CLONE_SYSVSEM	Parent and child share System V SEM_UNDO semantics.
CLONE_THREAD	Parent and child are in the same thread group.
CLONE_VFORK	vfork() was used and the parent will sleep until the child wakes it.
CLONE_UNTRACED	Do not let the tracing process force CLONE_PTRACE on the child.
CLONE_STOP	Start process in the TASK_STOPPED state.
CLONE_SETTLS	Create a new TLS (thread-local storage) for the child.
CLONE_CHILD_CLEARTID	Clear the TID in the child.
CLONE_CHILD_SETTID	Set the TID in the child.
CLONE_PARENT_SETTID	Set the TID in the parent.
CLONE_VM	Parent and child share address space.

Hebras kernel

- A veces es útil que el kernel realice operaciones en segundo plano, para lo cual se crean hebras kernel.
- Las hebras kernel no tienen un espacio de direcciones (su puntero mm es NULL)
- Se ejecutan únicamente en el espacio del kernel.
- Son planificadas y pueden ser expropiadas.
- Se crean por el kernel al levantar el sistema, mediante una llamada a `clone()`.
- Terminan cuando realizan una operación `do_exit` o cuando otra parte del kernel provoca su finalización.

Creación de procesos

`fork()` → `clone()` → `do_fork()` → `copy_process()`

Actuación de `copy_process`:

1. Crea la estructura thread_info (pila Kernel) y la task_struct para el nuevo proceso con los valores de la tarea actual.

2. Para los elementos de `task_struct` del hijo que deban tener valores distintos a los del padre, se les dan los valores iniciales correctos.

3. Se establece el estado del hijo a TASK_UNINTERRUPTIBLE mientras se realizan las restantes acciones.

Creación de procesos (y II)

4. Se establecen valores adecuados para los flags de la `task_struct` del hijo:

flag PF_SUPERPRIV = 0 (la tarea no usa privilegio de superusuario)

flag PF_FORKNOEXEC =1 (el proceso ha hecho fork pero no exec)

5. Se llama a `alloc_pid()` para asignar un PID a la nueva tarea.

6. Según cuáles sean los flags pasados a `clone()`, duplica o comparte recursos como archivos abiertos, información de sistemas de archivos, manejadores de señales, espacio de direccionamiento del proceso.....

7. Se establece el estado del hijo a TASK_RUNNING.

8. Finalmente `copy_process()` termina devolviendo un puntero a la `task_struct` del hijo.

Terminación de un proceso

- Cuando un proceso termina, el kernel libera todos sus recursos y notifica al padre su terminación.
- Normalmente un proceso termina cuando .....

  1) realiza la **llamada al sistema `exit()`;**

     De forma **explícita:** el programador incluyó esa llamada en el código del programa,

     O de forma **implícita:** el compilador incluye automáticamente una llamada a `exit()` cuando `main()` termina.

  2) **recibe una señal** ante la que tiene la acción establecida de terminar

- El trabajo de liberación lo hace la función **`do_exit()`** definida en `<linux/kernel/exit.c>`

Actuación de `do_exit()`

1. Activa el flag PF_EXITING de `task_struct`

2. Para cada recurso que esté utilizando el proceso, se decrementa el contador correspondiente que indica el nº de procesos que lo están utilizando

si vale 0 → se realiza la operación de destrucción oportuna sobre el recurso, por ejemplo si fuera una zona de memoria, se liberaría.

3. El valor que se pasa como argumento a exit() se almacena en el campo exit_code de task_struct (información de terminación para el padre)

4. Se manda una **señal al padre** indicando la finalización de su hijo.

Actuación de `do_exit()`

5. **Si aún tiene hijos**, se pone como padre de éstos al proceso init (PID=1)

(dependiendo de las características del grupo de procesos al que pertenezca el proceso, podría ponerse como padre a otro miembro de ese grupo de procesos)

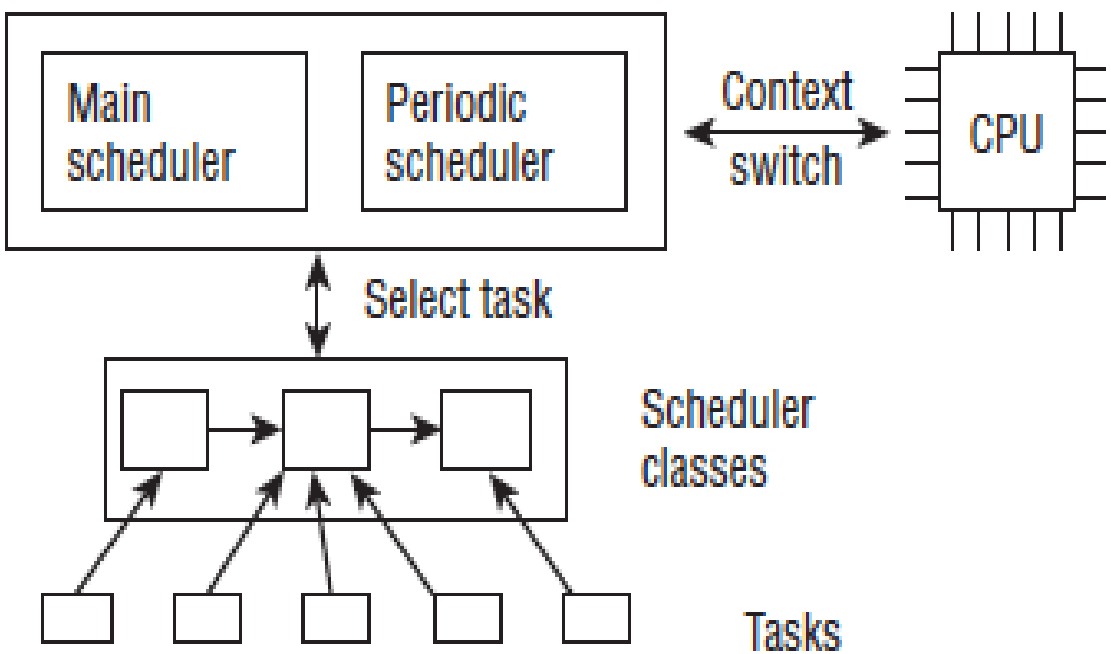
6. Se establece el campo exit_state de task_struct to EXIT_ZOMBIE

7. Se llama a `schedule()` para que el planificador elija un nuevo proceso a ejecutar

Puesto que este es el último código que ejecuta un proceso, do_exit nunca retorna

Planificación de la CPU en Linux

- Planificador modular: clases de planificación
  1. Planificación de tiempo real
  2. Planificación neutra o limpia (CFS: *Completely Fair Scheduling*)
  3. Planificación de la tarea "idle" (no hay trabajo que realizar)



Planificación de la CPU en Linux (y II)

- Cada clase de planificación tiene una prioridad
- Se usa un algoritmo de planificación entre las clases de planificación por prioridades apropiativo
- Cada clase de planificación usa una o varias políticas para gestionar sus procesos
- La planificación no opera únicamente sobre el concepto de proceso, sino que maneja conceptos más amplios en el sentido de manejar grupos de procesos: Entidad de planificación.
- Una entidad de planificación se representa mediante una instancia de la estructura `sched_entity`

Política de planificación

`unsigned int policy;` // política que se aplica al proceso

Políticas manejadas por el planificador CFS – `fair_sched_class:`

- `SCHED_NORMAL`: se aplica a los procesos normales de tiempo compartido
- `SCHED_BATCH`: tareas menos importantes, menor prioridad. Son procesos batch con gran proporción de uso de CPU para cálculos.
- `SCHED_IDLE`: tareas de este tipo tienen una prioridad mínima para ser elegidas para asignación de CPU

Políticas manejadas por el planificador de tiempo real – `rt_sched_class:`

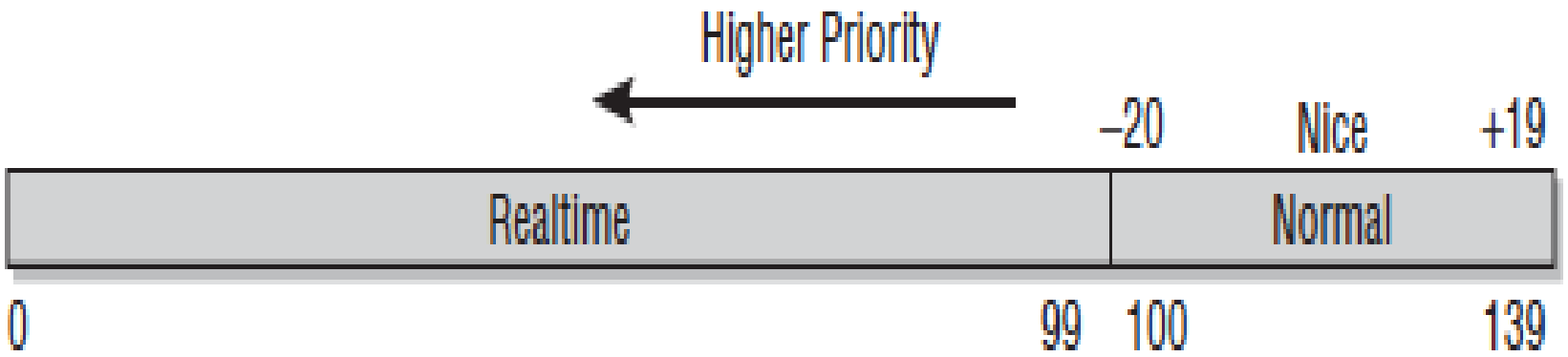
- `SCHED_RR`: uso de una política Round-Robin (Por turnos)
- `SCHED_FIFO`: uso de una política FCFS

Prioridades

- Siempre se cumple que el proceso que está en ejecución es el más prioritario.
- Rango de valores de prioridad para static_prio:

  [0, 99] Prioridades para procesos de **tiempo real**

  [100, 139] Prioridades para los procesos normales o regulares.



El planificador periódico

- Se implementa en `scheduler_tick`, función llamada automáticamente por el kernel con frecuencia HZ (constante cuyos valores están normalmente en el rango 1000 y 100Hz)
- Tareas principales:
  - actualizar estadísticas del kernel
  - activar el método de planificación periódico de la clase de planificación a que corresponde el proceso actual (`task_tick`). Cada clase de planificación tiene implementada su propia función `task_tick` (contabiliza en tº de CPU cosumido)
- Si hay que replanificar, el planificador de la clase concreta activará el flag `TIF_NEED_RESCHED` asociado al proceso en su `thread_info`, y provocará que se llame al planificador principal.

El planificador principal

- Se implementa en la función **`schedule`**, invocada en diversos puntos del kernel para tomar decisiones sobre asignación de la CPU.
- La función `schedule` es invocada de forma explícita cuando un proceso se bloquea o termina.
- El kernel chequea el flag `TIF_NEED_RESCHED` del proceso actual al volver al espacio de usuario desde modo kernel (ya sea al volver de una llamada al sistema o en el retorno de una interrupción o excepción y si está activo se invoca al schedule)

Actuación del `shedule`

- Determina la actual `runqueue` y establece el puntero `prev` a la `task_struct` del proceso actual
- Actualiza estadísticas y limpia el flag `TIF_NEED_RESCHED`
- Si el proceso actual estaba en un estado `TASK_INTERRUPTIBLE` y ha recibido la señal que esperaba, se establece su estado a `TASK_RUNNING`
- Se llama a `pick_next_task` de la clase de planificación a la que pertenezca el proceso actual para que se seleccione el siguiente proceso a ejecutar, se establece `next` con el puntero a la `task_struct` de dicho proceso seleccionado.
- Si hay cambio en la asignación de CPU, se realiza el cambio de contexto llamando a `context_switch`

La clase de planificación CFS

- Idea general: repartir el tiempo de CPU de forma imparcial, garantizando que todos los procesos se ejecutarán y, dependiendo del número de procesos, asignarles más o menos tiempo de uso de CPU.
- Mantiene datos sobre los tiempos consumidos por los procesos.
- El kernel calcula un peso para cada proceso. Cuanto mayor sea el valor de la prioridad estática de un proceso, menor será el peso que tenga.
- **vruntime** (virtual runtime) de una entidad es el *tiempo virtual* que un proceso ha consumido y se calcula a partir del tiempo real que el proceso ha hecho uso de la CPU, su prioridad estática y su peso.

La clase de planificación CFS (y II)

- El valor vruntime del proceso actual se actualiza:
  - periódicamente (el planificador periódico ajusta los valores de tiempo de CPU consumido)
  - cuando llega un nuevo proceso ejecutable
  - cuando el proceso actual se bloquea
- Cuando se decide qué proceso ejecutar a continuación, se elige el que tenga un valor menor de vruntime.
- Para realizar esto CFS utiliza un rbtree (red black tree):

  estructura de datos que almacena nodos identificados por una clave y que permite una eficiente búsqueda dada una determinado valor de la clave.

La clase de planificación CFS (y III)

- Cuando un proceso va a entrar en estado bloqueado:
  - se añade a una cola asociada con la fuente del bloqueo
  - se establece el estado del proceso a `TASK_INTERRUPTIBLE` o a `TASK_NONINTERRUPTIBLE` según sea conveniente
  - es eliminado del rbtree de procesos ejecutables
  - se llama a `schedule` para que se elija un nuevo proceso a ejecutar
- Cuando un proceso vuelve del estado bloqueado
  - se cambia su estado a ejecutable (`TASK_RUNNING`)
  - se elimina de la cola de bloqueo en que estaba
  - se añade al rbtree de procesos ejecutables

La clase de planificación de tiempo real

- Se define la clase de planificación `rt_sched_class`.
- Los procesos de tiempo real son más prioritarios que los normales, y mientras existan procesos de tiempo real ejecutables éstos serán elegidos frente a los normales.
- Un proceso de tiempo real queda determinado por la prioridad que tiene cuando se crea, el kernel no incrementa o disminuye su prioridad en función de su comportamiento.
- Las políticas de planificación de tiempo real `SCHED_RR` y `SCHED_FIFO` posibilitan que el kernel Linux pueda tener un comportamiento *soft real-time* (Sistemas de tiempo real no estricto).
- Al crear el proceso también se especifica la política bajo la cual se va a planificar. Existe una llamada al sistema para cambiar la política asignada.

Particularidades en SMP

- Para realizar correctamente la planificación en un entorno SMP (multiprocesador), el kernel deberá tener en cuenta:
  - Se debe repartir equilibradamente la carga entre las distintas CPUs
  - Se debe tener en cuenta la afinidad de una tarea con una determinada CPU
  - El kernel debe ser capaz de migrar procesos de una CPU a otra (puede ser una operación costosa)
- Periódicamente una parte del kernel deberá comprobar que se da un equilibrio entre las cargas de trabajo de las distintas CPUs y si detecta que una tiene más procesos que otra, reequilibra pasando procesos de una CPU a otra.