mma66
www.wuolah.com/student/mma66

Resumen TEMA 2 -ampliadoParte2.pdf

Resúmenes Tema1 y Tema2 -SO-

2º Sistemas Operativos

Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
UGR - Universidad de Granada

Resumen TEMA 2 –Procesos y Hebras

Conceptos fundamentales sobre procesos

CONCEPTO DE PROCESO

Un proceso es un programa en ejecución y la ejecución de este programa debe realizarse sin interferencias debidas a la ejecución de otros programas.

- Programa = fichero estático ejecutable
- Proceso = programa en ejecución = programa + estado de ejecución

La ejecución del programa puede caracterizarse por su **traza de ejecución**.
Para ejecutar un programa, un proceso requiere recursos del SO: memoria, CPU, etc.

CONCEPTOS SOBRE KERNEL Y PROGRAMA DE USUARIO

Con respecto a la ejecución en CPU se utilizan dos modos de ejecución: user y kernel.

Con respecto a la memoria:

●Espacio protegido para el kernel:
- La CPU opera sobre este espacio solamente en modo kernel.
- Se requiere una secuencia especial de instrucciones para cambiar de modo user a modo kernel.
- El código de este espacio es **reentrante**.

●Cada proceso tiene su propio espacio de direcciones protegido:
- Es necesario tener información sobre el estado de la memoria asociada a cada proceso. Esta información se guarda en espacio de memoria del kernel.
- Todos los procesos comparten el mismo espacio de kernel.
- Una pila de kernel por cada proceso.

MODO DE EJECUCIÓN, ESPACIO DE DIRECCIONES Y CONTEXTO

Modo / Contexto	User	Kernel
Proceso	Código de usuario	Llamadas al sistema y excepciones
Kernel	(No permitido)	Tratamiento de Interrupciones, Tareas del sistema.

EJECUCIÓN DEL SO

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

En el sistema hay muchos procesos simultáneamente. Cuando el SO decide que un proceso ejecutándose en CPU debe abandonarla, tiene que salvar los valores actuales de los registros de CPU (contexto de registros) en el **PCB** de dicho proceso.
La acción de conmutar la CPU de un proceso a otro se denomina cambio de contexto (***context switch***).

nocilla®

SIN ACEITE DE PALMA

LA MEJOR RECOMPENSA DESPUÉS DE UNA TARDE DE ESTUDIO

¿NOCILLEAMOS?

nociyeah!

PCB

Estructura de datos que contiene la información relativa al concepto de proceso. El PCB es creado, gestionado y destruido por el kernel.

● ***Información básica que contiene:***

– *Process IDentifier* (PID).
– *Process State* (*state diagram*).
– Contexto de Registros.
– Información de memoria.
– Lista de recursos utilizados

Un proceso tiene texto y datos asociados. Los "datos" (texto, datos y pila) y los metadatos (PCB) residen en memoria.
Un proceso tiene una pila asociada para poder realizar llamada a funciones.
El SO almacena el estado de ejecución del programa en el PCB.

CAMBIO DE CONTEXTO

Cuando un proceso está ejecutándose, los registros contienen el estado actual de la computación. Cuando el SO detiene un proceso que está ejecutándose, salva los valores actuales de estos registros en el PCB de dicho proceso.
La acción de cambiar al proceso que está usando la CPU por otro proceso se denomina **cambio de contexto**.

¿Cuándo se produce un cambio de contexto?

– El proceso en ejecución ha agotado su *time slice*.
– Como consecuencia de una llamada al sistema bloqueante.
– ¡Como consecuencia de una interrupción!
– ¡El proceso decide por sí mismo abandonar la CPU!

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.

● ¿Por qué motivos se puede crear un proceso?

– En sistemas *batch*: en respuesta a la recepción y admisión de un trabajo.
– "*logon*" interactivo. Cuando el usuario se autentifica desde un terminal (*logs on*), el SO crea un proceso que ejecuta el intérprete de ordenes asignado.
– 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 (relación padre-hijo).

¿Cómo funciona en UNIX-like OS?

- La llamada al sistema **fork()** crea un nuevo proceso (hijo).
- La llamada **exec()** reemplaza el espacio de direcciones con el programa pasado como argumento.
- Si se realiza una llamada **fork()** seguida de una llamada **exec()** se consigue un proceso hijo que ejecuta el programa pasado como argumento en **exec()**.

● **fork()** crea un nuevo hijo que hereda:
- Una copia idéntica de la memoria del padre.

- Una copia idéntica de los registros de CPU del padre(menos uno).

Los procesos padre e hijo se ejecutan en el mismo punto de ejecución: tras el retorno de **fork()** y, por convenio:

- **fork()** devuelve un 0 en el proceso hijo.
- **fork()** devuelve el identificador de proceso del hijo en el proceso padre.

Pasos fundamentales a realizar por un kernel cuando se solicita la operación de creación de proceso:

- Identificar al proceso asignándole un PID.
- Asignar espacio en memoria RAM y/o en memoria secundaria (SWAP) para el programa que se ejecutará.
- Crear el PCB e inicializar los campos de información.
- Insertar el PCB en la Tabla de Procesos y enlazarlo en la cola de planificación correspondiente, en caso de residir el programa en RAM.

¿Qué situaciones determinan la finalización de un proceso?

● Cuando un proceso ejecuta la última instrucción, solicita al SO su finalización mediante la llamada **exit()**, lo que implica:

- Aviso de finalización al padre (SIGCHLD) y guardar estado de finalización.
- Recursos asociados al proceso son liberados por el SO.

● El proceso padre puede finalizar la ejecución de sus procesos hijos mediante la llamada **kill()**.

● (Terminación en cascada) El proceso padre va a finalizar y el SO no permite a los procesos hijos continuar con la ejecución.

Threads (Hebras o hilos)

Una hebra es la unidad básica de utilización de la CPU y el kernel requiere la siguiente información para gestionarla:

- Contexto de registros.
- Pila de ejecución.
- Estado (diagrama de estados).

Single threading: Kernel no reconoce el concepto de hebra.

Multithreading: Kernel soporta multiples hebras dentro de un proceso.

VENTAJAS

-Se reduce el tiempo de cambio de contexto entre hebras, así como el tiempo de creación y terminación de hebras.

-Mientras una hebra de una tarea está bloqueada, otra hebra de la misma tarea puede ejecutarse, siempre que el kernel sea *multithreading*.

FUNCIONALIDAD

Al igual que los procesos las hebras poseen un estado (diagrama de estados) y pueden sincronizarse.

● *Estados de las hebras:*

- Nuevo, Ejecución, Listo, Bloqueado y Finalizado.

● *Operaciones básicas relacionadas con el cambio de estado en hebras:*

- Creación, Bloqueo, Desbloqueo y Terminación.

TIPOS DE HEBRAS ULT vs KLT

ULT

-Toda la gestión de hebras se realiza a nivel usuario.
-El kernel no es consciente de las actividades relacionadas con hebras ya que solo gestiona el proceso (tarea).
-La entidad de planificación para el kernel es el proceso.

KLT

-Toda la gestión de hebras se realiza a nivel kernel.
- El SO proporcional un conjunto de llamadas al sistema para la gestión de hebras.
- Muchas de las funciones que realiza el kernel son gestionadas mediante hebras kernel.
- La entidad de planificación para el kernel es la hebra.

ULT	KLT
Ventajas	**Ventajas**
El cambio de hebra no provoca cambio de modo.	El kernel puede planificar distintas hebras de la misma tarea en distintos procesadores.
La planificación de hebras puede adaptarse a las necesidades de la aplicación.	El bloqueo de una hebra de una tarea no provoca el bloqueo del resto de “hebras pares”.
Las aplicaciones se pueden ejecutar en cualquier SO.	Las rutinas del kernel pueden ser multihebra.
Desventajas	**Desventajas**
La mayoría de las llamadas al sistema son bloqueantes, por lo que si una hebra realiza una llamada, el resto de hebras se bloqueara.	El cambio de hebras dentro de la misma tarea se realiza en modo kernel, por lo que provoca un cambio de modo.
El kernel solo asigna procesos a procesadores, por lo que no se puede asignar más de un procesador a más de una hebra de la misma tarea.	

Conceptos fundamentales sobre planificación

MODELO GENÉRICO DE COLAS

El SO tiene una cola por cada estado. Cada PCB esta encolado en una cola de estado acorde a su estado actual, si su estado cambia se encola en otro sitio.

1- **Cola de trabajos**. Conjunto de los trabajos pendientes de ser admitidos en el sistema.

2- **Cola de LISTOS (Preparados para Ejecutar)**. Conjunto de todos los procesos, cuyos programas asociados residen en memoria principal, esperando para poder ejecutarse en la CPU.

3- **Colas de BLOQUEADOS**. Conjunto de todos los procesos esperando por un dispositivo de E/S particular o por un evento que debe producirse para poder continuar su ejecución.

TIPOS DE PLANIFICADORES

Planificador → Parte del SO que controla la utilización de un recurso.

- ***Tipos de planificadores:***
  - **Planificador a Largo plazo** (**Planificador de trabajos**)**:** selecciona los trabajos que deben llevarse a la cola de preparados. Permite controlar el grado de multiprogramación. Se invoca poco frecuentemente.
  - **Planificador a corto plazo** (**Planificador de CPU**): Trabaja con la cola de LISTOS, selecciona el proceso que debe ejecutarse a continuación, y le asigna la CPU. Se invoca muy frecuentemente.
  - **Planificador a Medio plazo.** FALTA ALGO QUE PONER DE DEFINICION

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 periodos 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.

DESPACHADOR (dispatcher)

Despachador: función del SO que da el control de la CPU al proceso seleccionado por el planificador a corto plazo. Esto involucra:

- Cambio de contexto de registros desde el punto de vista de la CPU (se realiza en modo kernel).
- Conmutación a modo usuario o a modo kernel dependiendo del nuevo contexto de registros.

Latencia de despacho. Tiempo que emplea el despachador en detener un proceso y comenzar a ejecutar otro.

MEDIDAS

Objetivos: buen rendimiento y buen servicio. Proceso→**P** - Tº de servicio→**r** (ráfaga).

- ***Tiempo de respuesta (T)***→ tiempo transcurrido desde que se remite una solicitud (entra en la cola de listos) hasta que se produce la primera respuesta.
- ***Tiempo de espera (M)***→ tiempo que un proceso ha estado esperando en la cola de listos (preparados): **T- r**
- ***Penalización (P)*** → **T/ r**
- ***Índice de respuesta (R)*** → Tiempo que P está recibiendo servicio. **r / T**

TIPOS DE POLÍTICAS DE PLANIFICACIÓN

≫ **No apropiativas (no expulsivas) (non-preemtive schedulers)**: una vez que se le asigna el procesador a un proceso, no se le puede retirar hasta que este voluntariamente lo deje (finalice o se bloquee).

≫ **Apropiativas (expulsivas) (preemptive)**: el SO puede apropiarse del procesador cuando lo decida.

REVISANDO context_switch()

context_switch() está compuesta de **schedule()**, planificador de CPU, y **dispatch()**, despachador.

● ¿Cuando llama el SO a **schedule()**? Es decir, ¿Cuándo toma decisiones de planificación el SO?

1. **E**jec → **B**loqueado, dentro de **RutinaE/S()** o **wait()** system call.
2. **E**jec → **F**inalizado, **sys_exit()** kernel algorithm.
3. **E**jec → **L**isto, Dentro de la **RSI_reloj()**.
4. {**N**uevo, **B**loqueado, **S**uspendido-**L**isto} → L, implementando CPU *preemption*.

Los planificadores no apropiativos (*non-preemtive schedulers*) solo usan 1 y 2.
Los planificadores apropiativos usan 1-4.

Políticas de planificación de la CPU

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 ejecutara hasta que finalice o se bloquee.
-Todos los procesos pierden la misma cantidad de tiempo esperando en la cola de ejecutables. Procesos cortos muy penalizados y procesos largos poco penalizados.

El más corto primero no apropiativo

Cuando el procesador queda libre, selecciona el proceso que requiera un **tiempo de servicio menor**. Si existen dos o más procesos con mismo tiempo, se sigue FCFS.
Necesita conocer explícitamente el tiempo de servicio estimado.
Disminuye el tiempo de respuesta para los procesos cortos y discrimina a los largos.

El más corto primero apropiativo

Cada vez que entra un proceso a la cola de listos se comprueba si su tiempo de servicio es menor que el tiempo de servicio que le queda al proceso que esta ejecutándose.

● **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**: continua el proceso que estaba ejecutándose.

El tiempo de respuesta es menor excepto para procesos muy largos.

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)
Se puede implementar con modalidades: Apropiativa y 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, RR*)

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 apropia y se coloca al final de la cola de ejecutables.

*Es **apropiativo***.

Penaliza a todos los procesos en la misma cantidad, sin importar si son cortos o largos. Las ráfagas muy cortas están mas penalizadas de lo deseable.

- Valor de quantum muy grande (excede del tiempo de servicio de todos los procesos) => se convierte en FCFS.
- Valor de quantum muy pequeño => el sistema acapara la CPU haciendo cambios de contexto (tiempo del núcleo muy alto)

Colas multiples

La cola de listos se divide en varias colas y cada proceso es asignado permanentemente a una cola concreta. Ej: Dos colas: Procesos interactivos y procesos *batch*

Cada cola puede tener su propio algoritmo de planificación. Ej. Interactivos con RR y procesos batch con FCFS

- ***Requiere una planificación entre colas:***
  - **Planificación con prioridades fijas**.
  - **Tiempo compartido entre colas**. Cada cola obtiene cierto tiempo de CPU que debe repartir entre sus procesos.

Colas multiples con realimentacion.

Un proceso se puede mover entre las distintas colas.

- ***Requiere definir los siguientes parámetros:***
  - Numero de colas
  - Algoritmo de planificación para cada cola
  - Método utilizado para determinar cuándo 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

Planificación en sistemas de tiempo real

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:***
  - tº real duro (*hard rt*). La tarea debe cumplir su plazo límite.
  - tº real suave (*soft rt*). La tarea tiene un tiempo límite pero no es obligatorio cumplirlo.
  - Periódicas. Se sabe cada cuanto 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.

Resumen TEMA 2 (PARTE 2) –Procesos y Hebras

Implementación de proceso/hebra en Linux: *task*

- 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.

ESTADOS DE UNA TAREA TASK

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 estructura task hasta que el padre haga un wait (**EXIT_DEAD**)

IMPLEMENTACION DE HILOS (threads)

- Desde el punto de vista del kernel no hay distinción entre hebra y proceso: *task = process/thread*.
- 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 dependiendo del parámetro flags.

HEBRAS KERNEL (kernel threads)

A veces es útil que el kernel realice operaciones en segundo plano, para lo cual se crean hebras kernel. Se ejecutan únicamente en el espacio del kernel.
Las hebras kernel no tienen un espacio de direcciones (su puntero mm es NULL). Son planificadas y pueden ser expropiadas.

- Solo se pueden crear por una hebra kernel mediante:

```
#include <include/linux/kthread.h>
struct task_struct *kthread_create_on_node(int (*threadfn)(void *data),
void *data, int node, const char namefmt[], ...);
```

- Terminan cuando realizan una operación ***do_exit*** o cuando otra parte del kernel provoca su finalización.

CREACIÓN DE TAREAS (task)

fork()* → *clone()* → *do_fork()* → *copy_process()

-Actuación de copy_process():

1. Crea la estructura *thread_info* (**pila kernel**) y la *task_struct* para la nueva tarea con los valores de la tarea actual.

2. Asigna valores iniciales a los campos de la *task_struct* de la tarea hija que deban tener valores distintos a los de la tarea padre.

3. Se establece el campo estado de la tarea hija ***TASK_UNINTERRUPTIBLE*** mientras se realizan las restantes acciones.

4. Se establecen valores adecuados para los flags de la *task_struct* de la tarea hija:

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 cuales 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 de la tarea hija a ***TASK_RUNNING***.

8. Finalmente *copy_process()* termina devolviendo un puntero a *la task_struct* de la tarea hija.

TERMINACIÓN DE TAREAS

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 incluyo 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** que tiene la disposición por defecto de terminar al proceso (Term).

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 número de procesos que lo están utilizando.
Si contador==0 → se realiza la operación de destrucción oportuna sobre el recurso, por ejemplo si fuera una zona de memoria, se liberaría el descriptor de memoria (*mm_struct*) correspondiente.

3. El valor que se pasa como argumento a *exit()* se almacena en el campo *exit_code* de *task_struct* (Esta es la información de terminación para que el padre pueda hacer un *wait()* o *waitpid()* y recogerla)

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

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

6. Se establece el campo exit_state de task_struct a EXIT_ZOMBIE.

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

Nota: Puesto que este es el último código que ejecuta un proceso, do_exit() nunca retorna.

Planificación de CPU en Linux

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.*

POLÍTICAS 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

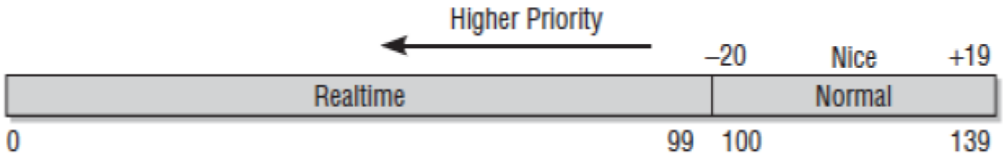
-SCHED_FIFO: uso de una política FCFS

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 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 el tiempo de CPU consumido).

- Si hay que replanificar, el planificador de la clase concreta activara el flag ***TIF_NEED_RESCHED*** asociado al proceso en su thread_info, y provocara que se llame al planificador principal.

EL PLANIFICADOR PRINCIPAL: Schedule()

- El planificador principal se implementa en la función *schedule(),* invocada en diversos puntos del kernel para tomar decisiones de 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 justo antes de volver de una llamada al sistema, de una rutina de servicio de interrupción (RSI) o de una rutina de servicio de excepción (RSE); si está activo este flag se invoca a *schedule().*

-Actuación de schedule ():

1. Determina la actual runqueue y establece el puntero previo a la *task_struct* del proceso actual.
2. Actualiza estadísticas y limpia el flag ***TIF_NEED_RESCHED***.
3. Si el proceso actual va a un estado ***TASK_INTERRUPTIBLE*** y ha recibido la señal que esperaba, se establece su estado a ***TASK_RUNNING***.
4. 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; y se establece next con el puntero a la *task_struct* de dicho proceso seleccionado.
5. Si hay cambio en la asignación de CPU, se realiza el cambio de contexto llamando a *context_switch()* (Nuestro dispatcher()).

LA CLASE DE PLANIFICACIÓN CFS

● **Idea general**: repartir el tiempo de CPU de forma imparcial, garantizando que todos los procesos se ejecutaran y, dependiendo del número de procesos en cola, asignarles más o menos tiempo de uso de CPU.
- 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.
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 tiene que decidir qué proceso ejecutar a continuación, se elige el que tenga un valor menor de vruntime.
- Para implementar esto CFS utiliza un red black tree (rbtree), que es una estructura de datos árbol binario que almacena nodos identificados por una clave, vruntime, y que permite una búsqueda eficiente por valor de clave.

Cuando un proceso va a entrar en estado bloqueado:
1. Se añade a una cola asociada con el motivo del bloqueo.
2. Se establece el estado del proceso a ***TASK_INTERRUPTIBLE*** o a ***TASK_NONINTERRUPTIBLE*** según este clasificado el motivo del bloqueo.
3. Se quita del rbtree de procesos ejecutables la referencia a la *task_struct* del proceso.
4. Se llama a schedule() para que se elija un nuevo proceso a ejecutar.

Cuando un proceso vuelve del estado bloqueado:
1. Se cambia su estado a ejecutable, TASK_RUNNING.
2. Se elimina de la cola de bloqueo en que estaba.
3. 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 estos 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 y existe una llamada al sistema para cambiar la política asignada.

PLANIFICACIÓN DE CPU 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.
● 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.