Resúmenes Tema 2

DGIIM

Introducción

Ejecución del S.O.

El sistema operativo se puede ejecutar de dos formas:

- Núcleo sin procesos. Es el modelo tradicional, las llamadas al sistema requieren guardar el contexto y pasar el control al kernel. En este caso, los procesos son solo los programas de usuario y el código del sistema operativo se ejecuta de forma independiente en modo *privilegiado*
- Como un proceso de usuario. El sistema operativo se ejecuta de forma virtual como si fuera un proceso. Si ocurre una interrupción, el procesador pasa a modo kernel(cambio de modo) pero no es necesario hacer un cambio de contexto, basta con guardar la información

Cambio de contexto

El cambio de contexto es la acción de cambiar la CPU de estar ejecutando un proceso a otro. Para esto, previamente se debe haber salvado la información del primer proceso(PC,SP, registros...) en su PCB para no perder esa información.

Procesos

Un proceso es un programa en ejecución.

1. Creación de un procesos. Al crear un proceso, se le asigna el espacio de direcciones que va a usar y se crean las estructuras de datos para administrarlo. Para crearlo verdaderamente, se realiza el siguiente proceso:

a) Se asigna un PID único.

b) Se reserva espacio para el proceso. (Para el espacio de direcciones privado y compartido, para la pila de usuario, PCB)

c) Se inicializa el PCB. Suele ser a 0 menos el PC y los punteros de pila del sistema. El proceso no debe poseer ningún recurso a menos que haya una indicación explícita o sea heredado del padre.

d) Se inserta en la tabla de procesos y se introduce en una cola de planificación.

e) Se crean otras estructuras de datos necesarias y se determina su prioridad.

Algunos casos comunes en los que se crea un proceso son: en respuesta a admisión y recepción de un trabajo(sistema batch), al conectar un usuario se crea un proceso que ejecuta el intérprete de órdenes(sistemas interactivos), para realizar un servicio solicitado por un proceso de usuario, o cuando un padre crea un proceso hijo.

Cuando un proceso crea a otro se deben tratar varios puntos. En cuanto a **recursos**, puede ocurrir que compartan todos los recursos, ninguno o un subconjunto. En cuanto a **ejecución**, ocurre que o bien se ejecutan a la vez o bien el padre espera a que el hijo termine. En cuanto al **espacio de direcciones** puede ser que el hijo sea un duplicado del padre o que el hijo tiene un programa que lo carga.

2. Terminación de procesos.

Puede ocurrir cuando:

a) El proceso solicita al SO su finalización. Entonces envía sus datos al padre y libera los recursos.

b) El padre aborta la ejecución de sus hijos si el hijo sobrepasa los recursos, su tarea ya no es necesaria o el padre va a finalizar

c) Lo aborta el SO pues se ha dado un fallo.

Hebras

Una **hebra** es una unidad básica de uso de la CPU. Contiene su PC, registros, espacio de pila y estado. Además, comparte con sus hebras pares una tarea que tiene su código, datos y recursos del SO.

Si un proceso es monohebra, le basta con su PCB, su espacio de direcciones y las pilas necesarias. Si es multihebra, necesita un PCB para cada hebra y las pilas para cada hebra.

Las hebras tienen como **ventajas** un mayor rendimiento y mejor servicio pues el tiempo de cambio de contexto, creación y terminación es menor, las hebras pueden ejecutarse de forma independiente y la comunicación entre hebras de la misma tarea se hace en memoria compartida.

Existen varios tipos de hebras:

- Hebras de usuario: La gestión la hace la aplicación, el núcleo no conoce que existen estas hebras. Son implementadas mediante una biblioteca en el nivel de usuario.
- Hebras kernel: Son gestionadas por el núcleo(que mantiene la información del contexto de todo el proceso y de cada hebra) y el SO proporciona llamadas a sistemas para trabajar con ellas. La unidad de planificación es la hebra y las funciones del núcleo pueden ser multihebras.

Las **ventajas** de las de tipo usuario frente a las de tipo núcleo son que no hay muchos cambios de modo, que existe una planificación para hebras distinta a la del SO y que se pueden ejecutar en cualquier SO sin hacer cambio en el nucleo.

Como **desventajas** tenemos que al bloquearse una hebra se bloquean todas las del proceso y que no se aprovechan las ventajas de un sistema multiprocesador, pues un proceso está asociado a un único procesador.

- Hebras híbridas(Hebras a nivel de kernel y de usuario): La creación y mayor parte de planificación y sincronización se hace en espacio de usuario. Las hebras de una aplicación se

relacionan con varias del núcleo y se pueden paralelizar para que si una hebra se bloquea no se bloquee todo el proceso.

Planificación

El SO tiene una colección de colas con el estado de todos los procesos. Suele haber una cola por estado. Al cambiar el estado de un proceso, su PCB se retira de una cola y se mete en otra.

El diagrama de estados por lotes es:

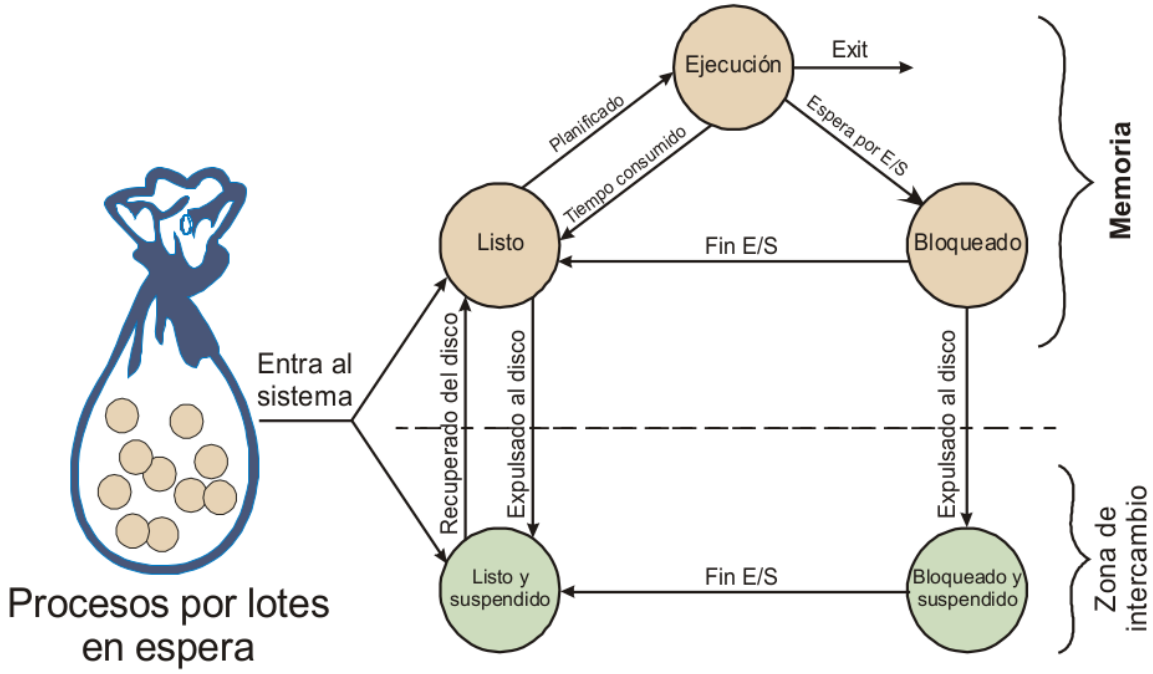

Figure 1: Diagrama de Estados

Colas de Estados

- Cola de trabajos: Pendientes de ser admitidos
- Cola de preparados: Residen en memoria principal esperando a ejecutarse.
- Cola(s) de bloqueados: Que esperan un suceso o una E/S

Planificador.

El planificador controla el uso de un recurso. Se encarga de asignar qué procesos van a ser ejecutados para cumplir los objetivos del sistema. Puede ser:

- **a largo plazo**, que selecciona los procesos que van a la cola de preparados (a memoria o a disco). Permite controlar el grado de multiprogramación y se invoca poco frecuentemente.

- **a corto plazo** que selecciona el siguiente proceso que se ejecuta y le asigna CPU. También bloquea los procesos en ejecución. Es el más frecuente. Trabaja con la cola de preparados.
- **a medio plazo** realiza una función de intercambio entre los de memoria y los de disco. Esto puede mejorar la mezcla de procesos o ayuda a cambiar los requisitos de memoria(realiza un intercambio o swapping).

Clasificación de procesos

- Limitados por E/S (o cortos): Dedican más tiempo a E/S que a cómputo, muchas ráfagas de CPU cortas y largos períodos de espera.
- Limitados por CPU(largos): Más tiempo en computación que en E/S. Al revés que los cortos.

El planificador a largo plazo debe hacer una buena mezcla de trabajos pues si no puede ocurrir que si todos los trabajos están limitados por E/S la cola de preparados estará casi siempre vacía y si todos los procesos están limitados por CPU la cola de E/S estará casi siempre vacía.

Dispatcher

El despachador es un módulo del SO que da el control de la CPU al proceso seleccionado por el planificador a corto plazo.Involucra cambio de contexto, cambio de modo a usuario y salto a la posición de memoria adecuada para la reanudación.

La latencia de despacho es el tiempo que tarda un despachador en cambiar de proceso ejecutándose.

El Despachador se activa cuando:

1. Un proceso no quiere seguir ejecutándose(acaba o se bloquea).
2. Un elemento del SO determina que el proceso no puede seguir ejecutándose(E/S)
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

Se pretende obtener buen rendimiento y servicio. Para saber si un proceso tiene un buen servicio definimos para un proceso P con tiempo de ráfaga(~quantum) t:

a) Tiempo de respuesta(T): tiempo desde que se remite una solicitud hasta que se produce la respuesta

b) Tiempo de espera(M): tiempo que ha estado un proceso en cola de preparados (T-t)

c) Penalización(P = T/t)

d) Índice de respuesta(R = t/T): tiempo que P está recibiendo servicio.

Otras medidas son tiempo del núcleo(tiempo perdido por el SO tomando decisiones de planificación), tiempo de inactividad (tiempo en el que la cola de ejecutables está vacía) y tiempo de retorno(tiempo necesario para ejecutar un proceso completo).

Clasificación de Políticas de planificación

1. No apropiativas(no expulsivas): Una vez que se asigna la CPU a un proceso no se le puede retirar hasta que éste se finalice o se bloquee
2. Apropiativas(expulsivas): El SO puede apropiarse del procesador cuando lo decida.

Los más importantes conocidos son:

- **FCFS**(First Come First Served). Los procesos se ejecutan según llegan a la cola de ejecutables. Es **no apropiativo**, cada proceso se ejecuta hasta que finalice o se bloquee. Es fácil de implementar pero pobre de prestaciones y todos los procesos pierden el mismo tiempo esperando en la cola de ejecutables. De esta forma, los procesos cortos se penalizan muchos y los largos poco.
- **SJF**(Shortest Job First). Se selecciona el proceso que requiere menos tiempo de CPU. Si hay dos procesos que tengan las mismas condiciones se sigue FCFS. Es **no apropiativo**. Se necesita saber el tiempo de ejecución estimado y disminiye el tiempo de espera para procesos cortos, los largos se ven afectados. El tiempo medio de espera es bajo.
- **SRTF**(El más corto primero apropiativo). Si un proceso entra a la cola de ejecutables se comprueba si su tiempo de servicio es menor que el tiempo de servicio restante al proceso ejecutándose. Si es menor, se cambia de contexto y se ejecuta y si no sigue el que estaba ejecutándose. El tiempo de respuesta es menor excepto para procesos largos y hay menos penalización en promedio.
- **Planificación por prioridades**. Se asocia a cada proceso un entero de prioridad. A la CPU se le asigna el proceso con mayor prioridad(enteros menores es más prioridad). Puede ser apropiativa o no apropiativa. El **problema** es que los procesos de baja prioridad pueden no ejecutarse nunca, pero como **solución** con el paso del tiempo se incrementa la prioridad de los procesos.
- **Round-Robin** (Por turnos). La CPU se asigna a los procesos en intervalos de tiempo(QUANTUMS). Si el proceso finaliza o se bloquea antes de agotar el quantum, se libera la CPU y se toma el siguiente proceso de la cola FIFO de ejecutables y se le asigna un quantum completo. Si el proceso no termina en su quantum, se interrumpe y se coloca al final de la cola de ejecutables (Es por tanto apropiativo). Este sistema penaliza a todos los procesos de la misma forma. Y las ráfagas muy cortas están más penalizadas. Si el valor del quantum es muy grande se convertirá en FCFS y si es muy pequeño el sistema monopoliza la CPU por los cambios de contexto.
- **Colas múltiples**. La cola de preparados se divide en varias colas y cada proceso se asigna permanentemente a una cola. Cada cola puede tener su algoritmo de planificación, pero debe existir una planificación entre colas (planificación con prioridades fijas o tiempo compartido).
- **Colas múltiples con realimentación**. Los procesos se pueden mover entre colas, pero requiere que definamos el número de colas, qué algoritmo de planificación hay en cada cola,un

método para saber cuándo hay que cambiar un proceso de cola, otro para determinar en qué cola se introducirá un proceso y un algorimo de planificación entre colas. Sin embargo, mide el tiempo de ejecución del comportamiento real de los procesos. Es la más general, se usa en Unix, Linux, Windows...

Tabla comparativa de los cuatro primeros:

	Apropiativo/no apropiativo	Tiempo respuesta	Efecto en procesos	Inanición	Otros
FSFC	No apropiativo	Alto si hay mucha diferencia entre los tiempos de ejecución	Penaliza procesos cortos y con E/S	No	Estado listo pasa a preparados
SJF	No apropiativo	Buen tiempo para procesos cortos. Discrimina los largos	Penaliza procesos largos	Posible	En caso de igualdad se usa FCFS
SRTF	Apropiativo	Buen tiempo excepto procesos muy largos	Penaliza procesos largos	Posible	
Por prioridad	Puede ser ambas		No se ejecutan los de prioridad baja	Sí	
Round Robin	Apropiativo	Buen tiempo para procesos cortos	Equitativo	No	Expulsión basada en quantum

Planificación en multiprocesadores

Se estudian tres aspectos relacionados: Asignación de procesos a procesadores(Cola para cada procesador y cola para todos los procesadores), uso de multiprogramación en cada procesador y activación del proceso.

- Planificación de procesos: Igual que en monoprocesadores pero se tiene en cuenta el número de CPUs y la asignación y liberación de proceso y procesador.
- Planificación de Hilos: para explotar el paralelismo dentro de una aplicación. En multiprocesadores existen varias formas.

a) Compartición de carga: Hay una cola global de hilos preparados y si un procesador está muy ocioso, se selecciona un hilo de la cola.

b) Planificación en pandilla: Se planifican varios hilos de un mismo proceso para ejecutarse dentro de un conjunto de procesadores al mismo tiempo. Es útil cuando los hilos necesitan sincronizarse.

c) Asignación de procesador dedicado: Se asigna un procesador a cada hilo hasta que termine la aplicación, pero puede que ocurra que haya procesadores vacíos(no hay multiprogramación).

d) Planificación dinámica: La aplicación permite que varíe dinámicamente el número de hilos de un proceso y el SO ajusta la carga para usar mejor los procesadores.

- Planificación de sistemas de tiempo real. Se enfoca según cuándo el sistema realice un análisis de viabilidad de la planificación (si puede atender a todos los eventos en su tiempo), si se realiza estática o dinámicamente o si el resultado del análisis produce un plan de planificación o no. Se utilizan enfoques estáticos dirigidos por una tabla(planificación que determina cuándo empezará cada tarea), estáticos expulsivos dirigidos por prioridad (sólo da prioridad a las tareas, no genera una planificación), enfoques dinámicos basados en plan(determina la viabilidad en tiempo de ejecución y se acepta si se pueden satisfacer sus restricciones de tiempo) y enfoques dinámicos de menor esfuerzo(sin análisis de viabilidad, se intenta cumplir los plazos y si no se cumple se aborta el proceso).

Problema de inversión de prioridad

Este problema ocurre en los planificadores con prioridad cuando una tarea de mayor prioridad espera por una tarea de menor prioridad porque hay un recurso bloqueado de uso exclusivo.

Para evitar el problema, se puede:

1. Establecer una herencia de prioridad (la tarea menos prioritaria hereda la prioridad de la más prioritaria)
2. Establecer un techo de prioridad (al proceso que se le asigna el recurso de uso exclusivo se le da una prioridad más alta)

En ambos, la 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 en el kernel 2.6 de Linux.

1. El núcleo identifica a los procesos por su PID
2. En Linux, proceso es la entidad que se crea con la llamada al sistema *fork* (excepto el proceso 0) y *clone*.
3. 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.

El kernel almacena la lista de procesos como una lista circular doblemente enlazada (task list). Cada elemento es un descriptor de un proceso (PCB) definido en

```
struct task_struct { /// del kernel 2.6.24
 volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
 /*...*/
/* Información para planificación */
 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;
/*...*/
//* 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 */
/* Información para planificación 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_estruct especifica el estado actual de un proceso.

1. **Ejecución** (TASK_RUNNING): Se corresponde con dos: ejecutándose o preparado para ejecutarse (en la cola de procesos preparados).
2. **Interrumpible** (TASK_INTERRUPTIBLE): El proceso está bloqueado, sale de este estado cuando ocurre el suceso por el cual está bloqueado o porque le llegue una señal.
3. **No interrumpible** (TASK_UNINTERRUPTIBLE): El proceso está bloqueado y sólo cambiará de estado cuando ocurra el suceso que esta esperando (no acepta señales).
4. **Parado** (TASK_STOPPED): El proceso ha sido detenido y sólo puede reanudarse por la acción de otro proceso (por ejemplo, proceso parado mientras está siendo depurado).
5. (TASK TRACED): El proceso está siendo traceado por otro proceso.
6. **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

[Diagrama de estados]{diagrama.png}

El árbol de procesos.

Cada *task_struct* tiene un puntero:

1. A la *task_struct* de su padre: struct task_struct *parent
2. A una lista de hijos (llamada children): struct list_head children
3. A una lista de hermanos (llamada sibling): struct list_head sibling

[Arbol de procesos]{diagrama2.png}

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 ms, 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);
```

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. Los crea el kernel al levantar el sistema, mediante una llamada a *clone()*. Terminan cuando realizan una operacion 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_UNINTERRUPTINLE* mientras se realizan las restantes acciones.
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. Se 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` del `task_struct` del proceso.
2. Para cada recurso que esté utilizando el proceso, se decrementa el contador correspondiente que indica el número de procesos que lo está utilizando.

- Si vale 0, entonces 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.
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` a `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.

- Planificación 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).

- 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 de planificación 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

El planificador `CFS-fair_sched_class` maneja varias políticas: * `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 tipo idle 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. El rango de valores para `static_prio` es [0,139], donde [0,99] suelen ser prioridades para procesos de tiempo real y [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. Sus tareas principales son actualizar las estadísticas del kernel y activar el método de planificación periódico de la clase de planificación a que corresponde el proceso actual. Si hay que replanificar, el planificador de la clase concreta acttivará el flag `TIF_NEED_RESCHED` asociado al proceso en su `thread_info`, y provocará que se llame al planificador principal.

El planificador principal

Este planificador se implementa en la función `schedule`, invocada de forma explícita, cuando un proceso se bloquea o termina, en diversos puntos del kernel para tomar decisiones sobre asignación de la CPU.

El kernel chequea el flag `TIF_NEED_RESCHED` del proceso actual al volver al espacio de usuario desde modo kernel y si está activo se invoca al schedule.

Actuación del `schedule`

1. El `schedule`, determina la actual `runqueue` y establece el puntero `prev` a la `task_struct` del proceso actual.
2. Actualiza estadísticas y limpia el flag `TIF_NEED_RESCHED`.
3. Si el proceso actual estaba en un estado `TASK_INTERRUPTIBLE` y ha recibido la señal que esperaba, se establece su estado a `TASK_RUNNING`.
4. Llama a `pick_next_task` de la clase de planificación del proceso actual para que se seleccione el siguiente proceso a ejecutar, se establece `next` con el puntero a la `task_struct` de dicho proceso.
5. Si hay cambio en la asignación de CPU, se realiza el cambio de contexto llamando a `contex_switch`.

Clase de planificación CFS

Con esto se pretende repartir el tiempo de CPU para garantizar que todos los procesos se ejecutarán y asignarles un tiempo de CPU que dependa del número de procesos. Para cada proceso, el kernel calcula un peso: cuanto mayor sea la prioridad estática de un proceso, menor peso tendrá. * *vruntime* de una entidad es el tiempo virtual que un proceso ha consumido (se calcula con el tiempo de uso de CPU, prioridad y peso). Su valor se actualiza periódicamente, cuando llega un nuevo proceso o si se bloquea el proceso actual. Al elegir un proceso a ejecutar, se usa el que tenga menor vruntime y para ello CFS utiliza un red black tree (TDA que almacena nodos identificados por una clave para eficiente búsqueda).

- Si un proceso va a bloquearse se añade a una cola asociada con la fuente del bloqueo y se establece su estado a TASK_INTERRUPTIBLE o TASK_NONINTERRUPTIBLE según convenga. Se elimina del rbtree de procesos ejecutables y se llama a schedule para que elija un proceso a ejecutar.
- Si un proceso vuelve del estado bloqueado, se cambia su estado a ejecutable (TASK_RUNNING), se elimina de la cola de bloqueo y se añade al rbtree de procesos ejecutables.

Clase de planificación de tiempo real

Es una clase definida como *rt_sched_class*. Los procesos de tiempo real son más prioritarios que los normales. Los de tiempo real quedan determinados por la prioridad que tienen al crearse, nos e modifica. Gracias a la planificación de tiempo real *SCHED_RR* y *SCHED_FIFO* Linux puede ser un sistema de tiempo real no estricto (*soft real-time*). Al crear un proceso tambien se especifica la política de planificación, existe una llamada al sistema para cambiar la política asignada.

Particularidades en SMP

En un entorno multiprocesador, el kernel debe repartir la carga bien entre las CPUs, tener en cuenta la afinidad tarea-cpu y ser capaz de migrar procesos entre CPUs. Periódicamente, el kernel debe comprobar si la carga está en equilibrio.