LECCIÓN 6: PLANIFICACIÓN DE PROCESOS

LECCIÓN 6: PLANIFICACIÓN DE PROCESOS

6.1 Introducción a los procesos

Los procesos llevan a cabo tareas en el sistema operativo. Un programa es un conjunto de instrucciones de código máquina y datos guardados en disco en una imagen ejecutable y como tal, es una entidad pasiva; podemos pensar en un proceso como un programa de computador en acción.

Un proceso es una entidad dinámica, cambiando constantemente a medida que el procesador ejecuta las instrucciones de código máquina. Un proceso también incluye el contador de programa y todos los registros de la CPU, así como las pilas del proceso que contienen datos temporales como parámetros de las rutinas, direcciones de retorno y variables salvadas. El programa que se está ejecutando, o proceso, incluye toda la actividad en curso en el microprocesador. Linux es un sistema multiproceso. Los procesos son tareas independientes, cada una con sus propios derechos y responsabilidades.

Si un proceso se desploma, no hará que otros procesos en el sistema fallen también. Cada proceso se ejecuta en su propio espacio de dirección virtual y no puede haber interacciones con otros procesos excepto a través de mecanismos seguros gestionados por el núcleo.

Durante la vida de un proceso, éste hará uso de muchos recursos del sistema. Usará las CPUs del sistema para ejecutar sus instrucciones y la memoria física del sistema para albergar al propio proceso y a sus datos. El proceso abrirá y usará ficheros en los sistemas de ficheros y puede usar dispositivos del sistema directa o indirectamente. Linux debe llevar cuentas del proceso en sí y de los recursos de sistema que está usando de manera que pueda gestionar este y otros procesos justamente. No sería justo para los otros procesos del sistema que un proceso monopolizase la mayoría de la memoria física o las CPUs.

El recurso más preciado en el sistema es la CPU; normalmente sólo hay una. Linux es un sistema operativo multiproceso. Su objetivo es tener un proceso ejecutándose en cada CPU del sistema en todo momento, para maximizar la utilización de la CPU. Si hay más procesos que CPUs (y normalmente así es), el resto de los procesos tiene que esperar a que una CPU quede libre para que ellos ejecutarse. El multiproceso es una idea simple; un proceso se ejecuta hasta que tenga que esperar, normalmente por algún recurso del sistema; cuando obtenga dicho recurso, puede ejecutarse otra vez. En un sistema uniproceso, por ejemplo DOS, la CPU estaría simplemente esperando quieta, y el tiempo de espera se desaprovecharía. En un sistema multiproceso se mantienen muchos procesos en memoria al mismo tiempo. Cuando un proceso tiene que esperar, el sistema operativo le quita la CPU a ese proceso y se la da a otro proceso que se la merezca más. El **planificador** se encarga de elegir el proceso más apropiado para ejecutar a continuación. Linux usa varias estrategias de organización del tiempo de la CPU para asegurar un reparto justo.

PROCESOS DE LINUX

Para que Linux pueda gestionar los procesos en el sistema, cada proceso se representa por una estructura de datos **task_struct** (las tareas (task) y los procesos son términos intercambiables en Linux). El vector **task** es una lista de punteros a estructuras *task_struct* en el sistema. Esto quiere decir que el máximo número de procesos en el sistema está limitado por el tamaño del vector task; por defecto tiene 512 entradas. A medida que se crean procesos, se crean nuevas estructuras *task_struct* a partir de la memoria del sistema y se añaden al vector task. Para encontrar fácilmente el proceso en ejecución, hay un puntero (*current*) que apunta a este proceso.

Linux soporta procesos de tiempo real así como procesos normales. Estos procesos tienen que reaccionar muy rápidamente a sucesos externos (de ahí el término "tiempo real"') y reciben un trato diferente del planificador. La estructura *task_struct* es bastante grande y compleja, pero sus campos se pueden dividir en áreas funcionales:

State

(Estado) A medida que un proceso se ejecuta, su *estado* cambia según las circunstancias. Los procesos en Linux tienen los siguientes estados:

- **Running**

  (Preparado) El proceso se está ejecutando (es el proceso en curso en el sistema) o está listo para ejecutarse (está esperando a ser asignado a una de las CPUs del sistema).

- **Waiting**

  (Esperando) El proceso está esperando algún suceso o por algún recurso. Linux diferencia dos tipos de procesos; *interrumpibles* e *ininterrumpibles*. Los procesos en espera interrumpibles pueden ser interrumpidos por señales mientras que los ininterrumpibles dependen directamente de sucesos de hardware y no se pueden interrumpir en ningún caso.

- **Stopped**

  (Detenido) El proceso ha sido detenido, normalmente porque ha recibido una señal. Si se están depurando errores en un proceso, éste puede estar detenido.

Zombie

Es un proceso que ya ha terminado pero cuya estructura task_struct permanece aún en el vector task. Se suele mantener para que el padre pueda extraer información útil de él.

Información de la Planificación de Tiempo de la CPU

El planificador necesita esta información para hacer una decisión justa sobre qué proceso en el sistema se merece más ejecutarse a continuación.

Identificadores

Cada proceso en el sistema tiene un identificador de proceso. El identificador no es un índice en el vector *task*, es simplemente un número. Cada proceso también tiene identificadores de usuario y grupo, que se usan para controlar el acceso de este proceso a los ficheros y los dispositivos del sistema.

Enlaces

En un sistema de Linux ningún proceso es independiente de otro proceso. Cada proceso en el sistema, excepto el proceso inicial (*init*), tiene un proceso padre. Los procesos nuevos no se crean, se copian, o más bien *se clonan* de un proceso existente. Cada estructura *task_struct* que representa un proceso mantiene punteros a su proceso padre y sus hermanos (los procesos con el mismo proceso padre) así como a sus propios procesos hijos. Se pueden ver las relaciones entre los procesos en ejecución en un sistema Linux con la orden *pstree*:

```
init(1)-+-crond(98)
        |-emacs(387)
        |-gpm(146)
        |-inetd(110)
        |-kerneld(18)
        |-kflushd(2)
        |-klogd(87)
```

```
|-kswapd(3)
|-login(160)---bash(192)---emacs(225)
|-lpd(121)
|-mingetty(161)
|-mingetty(162)
|-mingetty(163)
|-mingetty(164)
|-login(403)---bash(404)---pstree(594)
|-sendmail(134)
|-syslogd(78)
`-update(166)
```

Además, todos los procesos en el sistema también están en una doble lista encadenada cuya raíz es la estructura *task_struct* del proceso *init*. Esta lista permite al núcleo de Linux observar cada proceso del sistema.

Tiempos y Temporizadores

El núcleo mantiene conocimiento de la hora de creación de los procesos así como el tiempo de CPU que consume a lo largo de su vida. En cada paso del reloj, el núcleo actualiza la cantidad de tiempo en *jiffies* que el proceso en curso ha usado en los modos sistema y usuario. Linux también soporta temporizadores *de intervalo* específicos a cada proceso; los procesos pueden usar llamadas del sistema para instalar temporizadores para enviarse señales a sí mismos cuando el temporizador acaba. Estos temporizadores pueden ser de una vez, o periódicos.

Sistema de Ficheros

Los procesos pueden abrir y cerrar ficheros y la estructura *task_struct* de un proceso contiene punteros a los descriptores de cada fichero.

Contexto Específico del Procesador

Un proceso se puede ver como la suma total del estado actual del sistema. Cuando un proceso se ejecuta, está utilizando los registros, las pilas, etc, del procesador. Todo esto es el contexto del procesador, y cuando se suspende un proceso, todo ese contenido específico de la CPU se tiene que salvar en la estructura *task_struct* del proceso. Cuando el planificador reinicia un proceso, su contexto se pasa a la CPU para seguir ejecutándose.

6.2 Planificación

Todos los procesos se ejecutan parcialmente en modo usuario y parcialmente en modo sistema. La manera como el hardware soporta estos modos varía, pero en

general hay un mecanismo seguro para pasar de modo usuario a modo sistema y viceversa. El modo usuario tiene muchos menos privilegios que el modo sistema. Cada vez que un proceso hace una llamada de sistema, cambia de modo usuario a modo sistema y sigue ejecutándose. Llegado este punto, el núcleo se está ejecutando por el proceso.

En Linux, un proceso no puede imponer su derecho sobre otro proceso que se esté ejecutando para ejecutarse él mismo. Cada proceso decide dejar la CPU que está usando cuando tiene que esperar un suceso en el sistema. Por ejemplo, un proceso puede estar esperando a leer un carácter de un fichero. Esta espera sucede dentro de la llamada de sistema, en modo sistema; el proceso utiliza una función de una biblioteca para abrir y leer el fichero y la función, a su vez, hace una llamada de sistema para leer bites del fichero abierto. En este caso el proceso en espera será suspendido y se elegirá a otro proceso para ejecutarse.

Los procesos siempre están haciendo llamadas de sistema y por esto necesitan esperar a menudo. Aún así, si un proceso se ejecuta hasta que tenga que esperar, puede ser que llegue a usar una cantidad de tiempo de CPU desproporcionada y por esta razón Linux usa planificación con derecho preferente. Usando esta técnica, se permite a cada proceso ejecutarse durante poco tiempo, 200 ms, y cuando ese tiempo ha pasado, otro proceso se selecciona para ejecutarse y el proceso original tiene que esperar un tiempo antes de ejecutarse otra vez. Esa pequeña cantidad de tiempo se conoce como una *porción de tiempo*. El *planificador* tiene que elegir el proceso que más merece ejecutarse entre todos los procesos que se pueden ejecutar en el sistema. Un proceso ejecutable es aquel que está esperando solamente a una CPU para ejecutarse, es decir, está en estado de Preparado (*Running*).

Linux usa un algoritmo razonablemente simple para planificar las prioridades y elegir un proceso entre los procesos que hay en el sistema. Cuando ha elegido un nuevo proceso para ejecutar, el planificador salva el estado del proceso en curso, los registros específicos del procesador y otros contextos en la estructura de datos *task_struct*. Luego restaura el estado del nuevo proceso (que también es específico a un procesador) para ejecutarlo y da control del sistema a ese proceso. Para que el planificador asigne el tiempo de la CPU justamente entre los procesos ejecutables en el sistema, el planificador mantiene cierta información en la estructura *task_struct* de cada proceso:

policy *(política)*

Esta es la política de planificación que se aplicará a este proceso. Hay dos tipos de procesos en Linux, normales y de tiempo real. Los procesos de tiempo real tienen una prioridad más alta que los otros. Si hay un proceso de tiempo real listo para ejecutarse, siempre se ejecutará primero. Los procesos de tiempo real pueden tener ***dos tipos de políticas***: *"round robin"* (en círculo) y *"first in first out"* (el primero en llegar es el primero en salir). En la planificación *"round robin"*, cada proceso de tiempo real ejecutable se ejecuta por turnos, y en la planificación *"first in, first out"* cada proceso ejecutable se ejecuta en el orden que están en la cola de ejecución y el orden no se cambia nunca.

priority *(prioridad)*

Esta es la prioridad estática que el planificador dará a este proceso. También es la cantidad de tiempo (en *jiffies*) que se permitirá ejecutar a este proceso una vez que sea su turno de ejecución. Se puede cambiar la prioridad de un proceso mediante una llamada de sistema y la orden *renice*.

rt_priority *(prioridad de tiempo real)*

Linux soporta procesos de tiempo real y estos tienen una prioridad más alta que todos los otros procesos en el sistema que no son de tiempo real. Este campo permite al planificador darle a cada proceso de tiempo real una prioridad relativa. La prioridad del proceso de tiempo real se puede alterar mediante llamadas de sistema.

counter *(contador)*

Esta es la cantidad de tiempo (en *jiffies*) que se permite ejecutar a este proceso. Se decrementa a cada paso de reloj y cuando el proceso se ejecuta y lo consume por completo se iguala a *priority*.

El planificador se ejecuta desde distintos puntos dentro del núcleo. Se ejecuta después de poner el proceso en curso en una cola de espera y también se puede ejecutar al finalizar una llamada de sistema, exactamente antes de que un proceso vuelva al modo usuario después de estar en modo sistema. También puede que el planificador se ejecute porque el temporizador del sistema haya puesto el contador *counter* del proceso en curso a cero. Cada vez que el planificador se ejecuta, hace lo siguiente:

Proceso en curso

El proceso en curso tiene que ser procesado antes de seleccionar a otro proceso para ejecutarlo.

Si la política de planificación del proceso en curso es *round robin* entonces el proceso se pone al final de la cola de ejecución.

Si la tarea es INTERRUMPIBLE y ha recibido una señal desde la última vez que se puso en la cola, entonces su estado pasa a ser RUNNING (Preparado).

Si el proceso en curso a consumido su tiempo, su estado pasa a ser RUNNING (Preparado).

Si el proceso en curso está RUNNING (Preparado), permanecerá en ese estado.

Los procesos que no estén ni RUNNING (Preparados) ni sean INTERRUMPIBLE se quitan de la cola de ejecución. Esto significa que no se les considerará para ejecución cuando el planificador busque un proceso para ejecutar.

Selección de un proceso

El planificador mira los procesos en la cola de ejecución para buscar el que más se merezca ejecutarse. Si hay algún proceso de tiempo real (aquellos que tienen una política de planificación de tiempo real) entonces éstos recibirán un mayor peso que los procesos ordinarios. El peso de un proceso normal es su contador *counter* pero para un proceso de tiempo real es su contador *counter* más 1000. Esto quiere decir que si hay algún proceso de tiempo real que se pueda ejecutar en el sistema, estos se ejecutarán antes que cualquier proceso normal. El proceso en curso, que ha consumido parte de su porción de tiempo (se ha decrementado su contador *counter*) está en desventaja si hay otros procesos con la misma prioridad en el sistema; esto es lo que se desea. Si varios procesos tienen la misma prioridad, se elige el más cercano al principio de la cola. El proceso en curso se pone al final de la cola de ejecución. En un sistema equilibrado con muchos procesos que tienen las mismas prioridades, todos se ejecutarán por turnos. Esto es lo que conoce como planificación *Round Robin* (en círculo). Sin embargo, como los procesos normalmente tienen que esperar a obtener algún recurso, el orden de ejecución tiende a verse alterado.

Cambiar procesos

Si el proceso más merecedor de ejecutarse no es el proceso en curso, entonces hay que suspenderlo y poner el nuevo proceso a ejecutarse. Cuando un proceso se está ejecutando, está usando los registros y la memoria física de la CPU y del sistema. Cada vez que el proceso llama a una rutina le pasa sus argumentos en registros y puede poner valores salvados en la pila, tales como la dirección a la que regresar en la rutina que hizo la llamada. Así, cuando el planificador se ejecuta, se ejecuta en el contexto del proceso en curso. Estará en un modo privilegiado (modo núcleo) pero aún así el proceso que se ejecuta es el proceso en curso. Cuando este proceso tiene que suspenderse, el estado de la máquina, incluyendo el contador de programa (program counter, PC) y todos los registros del procesador se salvan en la estructura *task_struct*. A continuación se carga en el procesador el estado del nuevo proceso. Esta operación es dependiente del sistema; diferentes CPUs llevan esta operación a cabo de maneras distintas, pero normalmente el hardware ayuda de alguna manera.

El cambio del contexto de los procesos se lleva a cabo al finalizar el planificador. Por lo tanto, el contexto guardado para el proceso anterior es una imagen instantánea del contexto del hardware del sistema tal y como lo veía ese proceso al final del planificador. Igualmente, cuando se carga el contexto del nuevo proceso, también será una imagen instantánea de cómo estaban las cosas cuando terminó el planificador, incluyendo el contador de programa (program counter, PC) de este proceso y los contenidos de los registros.

Es importante diferenciar entre el algoritmo de planificación propiamente dicho, y el cambio de contexto:

- El **algoritmo de planificación** únicamente tiene que decidir cuál será el siguiente proceso que utilizará el procesador. Se elige un proceso ganador entre todos los que no están bloqueados.

- El **cambio de contexto** es el mecanismo que efectivamente pone en ejecución a un proceso. Esta operación es muy dependiente de la arquitectura del procesador sobre el que se esté ejecutando, por lo que se tiene que realizar a bajo nivel (en ensamblador).

6.3 Representación de los procesos en el núcleo

La estructura de datos del núcleo que representa un proceso es ***struct task_struct*** definida en el fichero <*linux/shed.h*>. Linux utiliza un tabla de procesos compuesta por este tipo de estructuras llamada *task*. Cada proceso tiene asignado un identificador mediante el cual se obtiene su entrada correspondiente en la tabla.

Los campos más importantes de la estructura que representa un proceso son los siguientes:

Tipo	***Campo***	***Descripción***
struct_task struct *	next_task	Puntero al siguiente proceso en la lista
struct_task struct *	prev_task	Puntero al anterior proceso en la lista
struct_task struct *	next_run	Puntero al siguiente proceso en la lista de procesos Preparados
struct_task struct *	prev_run	Puntero al anterior proceso en la lista de procesos Preparados
struct_task struct *	p_pptr, p_opptr, p_ysptr, p_cptr, p_osptr	Punteros a los procesos padre, padre original, hermano mas reciente, hijo más reciente y hermano más antiguo
Int	pid	Identifiador del proceso
unsigned short	uid, euid, gid, egid, …	Identificadores de usuario (real y efectivo), grupo, ...
volatile long	state	Estado del proceso
Long	counter	Número de ciclos de reloj durante los que está autorizado a ejecutarse
unsigned short	timeout	Máximo tiempo de espera en estado de Espera
unsigned short	policy	Política de planificación asociada al proceso
Long	priority	Prioridad estática del proceso
unsigned short	rt_priority	Prioridad estática para procesos en tiempo real
unsigned long	signal	Señales en espera
struct signal_struct *	sig	Puntero a los descriptores de las acciones asociadas a las señales
unsigned long	blocked	Señales ocultas

Int	exit signal	Código de la señal a enviar al padre al finalizar
Int	exit code	Código a devolver al padre: bits(0..7): número de la señal que provocó el final del proceso bits(8..15): código devuelto por la primitiva *_exit*
struct wait_queue *	wait_chldexit	Variable utilizada para esperar la finalización de un proceso hijo
Int	errno	Variable global donde las llamadas al sistema colocan el código de error que generan
Long	utime, stime, cutime, cstime	Tiempos consumidos por el proceso y por sus hijos en modo usuario y modo núclo
int:1	dumpable	Indica si se debe crear un fichero CORE en caso de error fatal.
int:1	swappable	Indica si el proceso se puede guardar en memoria secundaria
long[8]	debugreg	Copia de los registros hardware para depuración
struct desc_struct *	ldt	Puntero a la tabla de segmentos local del proceso (LDT)
struct thread_struct *	tss	Puntero a una estructura donde se guarda el valor de los registros del procesador
struct mm_struct *	mm	Información de control para la gestión de la memoria
struct files_struct *	files	Puntero a los descriptores de ficheros abiertos por el proceso
struct fs_struct *	fs	Información de control para el acceso a ficheros
struct tty_struct *	tty	Puntero al terminal asociado al proceso

En el fichero <*linux/shed.h*> no sólo está declarada la estructura task sino que también están declaradas las estructuras que son direccionadas por algún campo de ésta (punteros) como por ejemplo la estructura *struct files_struct* y que están asociadas a diferentes partes del nucleo, como por ejemplo, el sistema de ficheros. Las definiciones de esas estructuras se han obviado para centrarnos en el concepto proceso y en su planificación.

El código C que define la estructura es el siguiente:

```
struct task_struct {
/* these are hardcoded - don't touch */
  /* -1 unrunnable, 0 runnable, >0 stopped */
```

```
                    /* Estado del proceso */
    volatile long state;

                    /* Flags */
    /* per process flags, defined below */
    unsigned long flags;
                    /* Señales pendientes */
    int sigpending;
                    /* Segmentos de memoria */
                    /* Espacio de direcciones para el hilo:
                     * 0-0xBFFFFFFF Para un hilo de usuario
                     * 0-0xFFFFFFFF Para un hilo del núcleo */
    mm_segment_t addr_limit;
    /* thread address space:
     * 0-0xBFFFFFFF for user-thead
     * 0-0xFFFFFFFF for kernel-thread */
    struct exec_domain *exec_domain;
                    /* necesita replanificarse */
    long need_resched;

  /* various fields */
                    /* Cuanto de tiempo restante */
    long counter;
                    /* prioridad Estatica */
    long priority;
    cycles_t avg_slice;
    /* Lock depth. We can context switch in and out of
     * holding a syscall kernel lock... */
                    /* Estructura de punteros entre tareas */
    struct task_struct *next_task, *prev_task;
    struct task_struct *next_run,  *prev_run;

          /* Punteros al proceso padre, al hijo más joven, al siguiente hermano más joven, al
          siguiente hermano más viejo */
    /* pointers to (original) parent process, youngest
     * child, younger sibling, older sibling, respectively.
     * (p->father can be replaced with p->p_pptr->pid) */
    struct task_struct *p_opptr, *p_pptr, *p_cptr,
      *p_ysptr, *p_osptr;

          /* Puntero al array task[] */
    /* Pointer to task[] array linkage. */
    struct task_struct **tarray_ptr;

          /* Politica aplicada y prioridad en tiempo real */
    unsigned long policy, rt_priority;

          /* Punteros a las estructuras asociadas a la tabla */
  /* tss for this task */
    struct thread_struct tss;
```

```
/* filesystem information */
  struct fs_struct *fs;
/* open file information */
  struct files_struct *files;
/* memory management info */
  struct mm_struct *mm;
};
```

La tabla *task* está definida como un array de punteros a variables del tipo *struct task_structs*. Cada tarea en el sistema está representado por una entrada en este array. El array es de tamaño NR_TASKS (definido a 512 en el fichero *"include/linux/task.h")* imponiendo el número máximo de tareas que pueden estar ejecutándose al mismo tiempo. Debido a que hay 32768 posibles PIDs, este array no es lo sufientemente grande para indexar directamente a todas las tareas en el sistema por sus PIDs. En vez de eso, Linux usa otras estructuras de datos para ayudarse a manejar estos recursos limitados. La estructura *task* es la siguiente:

```
/* Declaración de la tabla de procesos */
 extern struct task_struct *task[NR_TASKS];
```

La lista de slots libres, *tarray_freelist*, mantiene una lista (realmente una pila) de las posiciones libres en el array *task*. Este array se inicializa en la función *sched_init* y es manipulado por medio de dos funciones inline llamadas *add_free_taskslot* y *get_free_taskslot*. La estructura *tarray_freelist* es la siguiente:

```
/* Declaración de la lista de PID libres */
 extern struct task_struct ** tarray_freelist;
```

El array *pidhash* ayuda a mapear PIDs a punteros a estructuras del tipo *struct task_structs*. p*idhash* se inicializa en la función *sched_init*, posteriormente este array es manipulado con un conjunto de macros y funciones inline. Notar que las funciones que mantienen la estructura *pidhash* usan 2 miembros de la estructura *struct task_struct* que son *pidhash_next* y *pidhash_pprev*. Con esta estructura el kernel puede encontrar eficientemente una tarea por su PID.

```
 extern struct task_struct * pidhash[PIDHASH_SZ];
```

Las tareas forman una lista circular, doblemente encadenada llamada *lista de tareas*. Para llevarla a cabo se usan los campos *next_task* y *prev_task* de la estructura. Además de esta lista, todos los procesos se encuentran almacenados en el array *task*. Esta lista nos proporciona una manera rápida de movernos a través de todas las tareas actuales en el sistema evitándonos recorrernos todo el array *task* en el que pueden haber muchos slots vacíos. Para llevar a cabo una iteración a través de todas los tareas en el sistema, Linux proporciona una macro que nos facilita esta labor. Esta macro es la siguiente:

```
#define for_each_task(p)                        \
for(p = &init_task; (p = p->next_task) != &init_task; )
```

Esta macro se recorre la lista de tareas desde *init_task* a *init_task*. La tarea *init_task* no existe como tarea, sólo existe como marcador de principio y final en la cola circular. *init_task* no es visitada como parte de la iteración.

Además de esta lista, Linux guarda otra lista circular doblemente encadenada de tareas llamada *"run_queue"* (que normalmente se trata como una cola, de ahí su nombre), similar a la *lista de tareas*. Para implementar esta estructura se usan los campos "prev_run" y "next_run" de la estructura *"struct task_struct"*. Como ocurre con el campo *"next_task"* en campo *"prev_run"* sólo se necesita para eliminar efecientemente una entrada de la cola. La iteración a través de la lista se realiza siempre hacia delante utilizando el campo *"next_run"*, al igual que ocurra con la lista de tareas *"init_task"* se utiliza para indicar el comienzo y final de la cola circular.

Las tareas se añaden a la cola utilizando la función *"add_to_runqueue"* y se eliminan con *"del_from_runqueue"* definidas en <*linux/shed.c*>. A veces, es necesario mover procesos entre el principio y el final de la cola, utilizando las funciones *"move_first_runqueue"* y *"move_last_runqueue"* respectivamente.

La organización de los procesos puede verse como un grafo llamado Grafo de Procesos que expresa las relaciones "familiares" entre los procesos. El siguiente gráfico representa las relaciones "familiares" entre procesos:

La gestión de los enlaces entre procesos se realiza mediante las macros *"REMOVE_LINKS"* y *"SET_LINKS"* definidas en *"shed.h"*.

```
#define REMOVE_LINKS(p) do {                    \
  (p)->next_task->prev_task = (p)->prev_task;       \
  (p)->prev_task->next_task = (p)->next_task;       \
  if ((p)->p_osptr)                             \
    (p)->p_osptr->p_ysptr = (p)->p_ysptr;           \
  if ((p)->p_ysptr)                             \
    (p)->p_ysptr->p_osptr = (p)->p_osptr;           \
  else                                          \
    (p)->p_pptr->p_cptr = (p)->p_osptr;             \
  } while (0)
```

```
#define SET_LINKS(p) do {                       \
  (p)->next_task = &init_task;                  \
  (p)->prev_task = init_task.prev_task;             \
  init_task.prev_task->next_task = (p);             \
  init_task.prev_task = (p);                    \
  (p)->p_ysptr = NULL;                          \
  if (((p)->p_osptr = (p)->p_pptr->p_cptr) != NULL)     \
    (p)->p_osptr->p_ysptr = p;                      \
```

```
  (p)->p_pptr->p_cptr = p;                    \
  } while (0)
```

Otras líneas interesantes del fichero *<linux/shed.h>* son las siguientes:

```
extern int nr_running, nr_tasks;
extern int last_pid;


/* Definición de constantes */
#define TASK_RUNNING          0
#define TASK_INTERRUPTIBLE    1
#define TASK_UNINTERRUPTIBLE  2
#define TASK_ZOMBIE           4
#define TASK_STOPPED          8
#define TASK_SWAPPING         16

/* Scheduling policies */
#define SCHED_OTHER           0
#define SCHED_FIFO            1
#define SCHED_RR              2

/* This is an additional bit set when we want to yield
 * the CPU for one re-schedule..  */
#define SCHED_YIELD           0x10

#define MAX_SCHEDULE_TIMEOUT    LONG_MAX
```

En este fichero existen varias funciones que manejan la tabla de procesos y que no forman parte del planificador de tareas, pero que aparecen en este fichero porque la tabla está definida aquí, por ejemplo, las funciones *"add_free_taskslot"*, *"get_free_taskslot"* que son utilizadas por la primitiva *"fork()"*.

6.4 Planificación en linux

La planificación en Linux se lleva a cabo en la función ***schedule*** del fichero kernel/sched.c.

Linux implementa el esquema de planificación definido en las extensiones de tiempo real de POSIX (SCHED_FIFO y SCHED_RR son las políticas orientadas a procesos de tiempo real, como por ejemplo un grabador de CD-RW o un reproductor de música en MP3. Desde un programa, se elige la política de planificación mediante la llamada al sistema *sched_setscheduler*). Existen tres formas de planificar un proceso:

- *SCHED_OTHER*

Es la política de planificación "clásica" de UNIX (el proceso no es un proceso de tiempo real).
Este tipo de procesos sólo se puede ejecutar cuando no existe ningún proceso de tiempo real en estado Preparado.
El proceso a ejecutar se elige tras examinar las prioridades dinámicas. La prioridad dinámica de un proceso se basa, por una parte, en el nivel especificado por el usuario (utilizando las llamadas al sistema nice y setpriority; sólo un proceso privilegiado puede aumentar la prioridad, el resto, sólo puede bajarla) y por otra parte, en una variación calculada por el sistema. Todo proceso que se ejecute durante varios ciclos de reloj, disminuye en prioridad, pudiendo llegar a ser menos prioritario que los procesos que no se ejecutan, cuya prioridad no se ha modificado.

- *SCHED_FIFO*

  Es un proceso en tiempo real y está sujeto a un planificador FIFO (unix 4).
  Este se ejecutará hasta que:

  - se bloquee al necesitar esperar por la finalización de una operación de E/S
  - el procesador sea solicitado por otro proceso en tiempo real con una prioridad mayor
  - el proceso ceda voluntariamente el procesador utilizando la primitiva sched_yield.

  En la implementación Linux, los procesos SCHED-FIFO tienen una rodaja de tiempo asignada, aunque no podrán ser forzados a dejar la CPU cuando su tiempo se haya acabado, y por lo tanto, actúan como si no tuvieran rodaja de tiempo.

- *SCHED_RR*

  Significa que es un proceso en tiempo real sujeto a un planificador (unix 4) Round Robin.
  Funciona igual que un SCHED_FIFO excepto que las rodajas de tiempo si importan.
  Cuando a un proceso se le acaba su rodaja, el siguiente proceso al que se le asignará el procesador, se puede escoger de la lista de procesos SCHED_RR o de la lista de procesos SCHED_FIFO.

Existe además un bit adicional, SCHED_YIELD, que afecta a las tres políticas de planificación, haciendo que un proceso ceda la CPU a cualquier otro que se encuentre preparado. Este bit se activa mediante la llamada al sistema *sched_yield*, que activa la cesión de la CPU únicamente para el próximo ciclo de planificación, no indefinidamente.

ESQUEMAS DE PRIORIDAD EN LINUX

Cada proceso en Linux tiene una prioridad dada por un entero del 1 al 40, cuyo valor está almacenado en el campo *priority* de la estructra “struct task_struct”. Para procesos en tiempo real también se usa otro campo de dicha estructura llamado “rt_priority”.

Actualmente las funciones que controlan la prioridad son *sys_set_priority* y *sys_nice*, aunque esta última está obsoleta.

Los procesos en tiempo real de Linux añaden un nivel más al esquema de prioridad. La prioridad en tiempo real es llevada a cabo en el campo *rt_priority* de *struct task_struct*, y tiene un rango entero de entre 0 y 99 (un valor de 0 significa que el proceso no es un proceso de tiempo real, en el cual su campo *policy* debe ser *SCHED_OTHER*).

Las tareas en tiempo real usan el mismo campo *counter* como su homólogo en tiempo no real, por tanto su prioridad dinámica está manejada de la misma forma. Las tareas en tiempo real usan incluso el campo *priority* para los mismos propósitos que las tareas en tiempo no real, como el valor con el cual rellenar el campo *counter* cuando éste se agota. Para clarificarlo, *rt_priority* se usa sólo para ordenar los procesos en tiempo real por prioridad.

El campo *rt_priority* de un proceso se establece como parte de política de planificación con las funciones *sched_setsheduler* y *sched_setparam* (que sólo podrán ser llamadas por el root). Notar que esto significa que una política de planificación de procesos puede cambiar durante su tiempo de vida, asumiendo que esta tiene permiso para hacer el cambio.

Las llamadas al sistema sys_sched_setscheduler y sys_sched_setparam ambas delegan todo el trabajo real en setscheduler.

FUNCIÓN SCHEDULE

A continuación vamos a analizar paso a paso el código de *schedule*, está función representa el corazón de la planificación de procesos en Linux.

- Se definen al principio los punteros a la tabla de procesos prev y next, que apuntarán, respectivamente, al proceso que estaba en ejecución antes de planificar y al que se debe poner en marcha (prev y next pueden ser el mismo proceso).

- Si el proceso actual (prev) sigue la política *round robin*, entonces hay que comprobar si ya se le ha acabado su cuanto de tiempo y en su caso enviarlo al final de la cola de preparados.

- Si el proceso actual no está en el estado TASK_RUNNING, hay que quitarlo de la cola de preparados. Si está bloqueado pero aceptando señales

(TASK_INTERRUPTIBLE), se comprueba si hay que enviarle una señal y despertarlo. Si no, se elimina también de la cola (al no ejecutarse el break).

- Se elimina la marca de que hay que planificar (*need_resched*).

- La tarea *init* siempre está preparada (es la que se considera como "en ejecución" cuando no hay ningún proceso real preparado) y se toma como punto de entrada a la lista de procesos activos. Se recorrerá esta lista buscando el proceso más prioritario en ese momento, que quedará en next.

- El puntero a la tabla de procesos *p* va a ir recorriendo la lista. En *c* tendremos siempre el mayor valor de *goodness* (prioridad) obtenido.

- La función goodness devuelve un valor que representa la urgencia de ejecución de acuerdo a diversos criterios:

  - El valor base de weight (variable devuelta finalmente), resulta de sumar el numero de ticks que le quedan al proceso (p->counter, prioridad dinámica) más su prioridad estática (p->priority, valor típicamente asignado con nice(2)).

  - Los procesos de tiempo real (SCHED_FIFO y SCHED_RR) tienen más importancia que el resto de procesos. Su peso es 1000 más que el de los procesos "normales" (SCHED_OTHER).

  - Para reducir cambios de contexto, se le da una pequeña ventaja a los procesos que comparten el mismo mapa de memoria que el proceso actual (threads).

Se trata de una función crítica que ha alcanzado un equilibrio difícil de mejorar. Hay que tener en cuenta que la función goodness es invocada **varias veces (pueden ser muchas) cada vez que se planifica**. Esto implica que su simplicidad y eficiencia son cruciales para las prestaciones del sistema.

- Si el bucle no ha encontrado a ningún proceso con cuanto por ejecutar, entonces se llena el *cuanto* de todos los procesos. El contador de cada proceso se rellena con el valor anterior (que normalmente será cero, a menos que no hayamos llegado a entrar en goodness por tener activo el bit SCHED_YIELD) dividido por dos, más la prioridad estática. Esto significa que normalmente se reinicializan los contadores a la prioridad estática. El objetivo fundamental de esta división es evitar que en sucesivas invocaciones de la rutina *recalculate* (función llamada cuando se necesita rellenar el cuanto de tiempo de todos los procesos), un proceso que no está en ejecución aumente de forma indefinida su prioridad. De esta forma se impide que un proceso normal (SCHED_OTHER) supere la prioridad de los procesos de tiempo real.

  Obsérvese cómo a todos los procesos se les actualiza su variable *counter* aunque no estén en la cola de procesos activos. De esta forma se beneficia a los procesos que necesitan poco tiempo de CPU (procesos interactivos como

editores de texto, intérpretes de órdenes, etc) y se penaliza a los que consumen mucha.

- Si el proceso candidato (el más prioritario) no es el mismo que el último proceso en ejecución se producirá un cambio de contexto.

- Se incrementa el número de cambios de contexto. Dato que se puede consultar en el fichero especial *"/proc/stat"*

  switch_to es una macro que se expande a código ensamblador, desde donde se realiza el cambio de contexto.

  De **switch_to no se vuelve** (de ahí se pasará a ejecutar el próximo proceso) (al menos hasta que "nos vuelvan a planificar").

El código de la función *schedule* es el siguiente:

```
/* 'schedule()' is the scheduler function. It's a very
 * simple and nice scheduler: it's not perfect, but
 * certainly works for most things.
 *
 * The goto is "interesting".
 *
 * NOTE!!  Task 0 is the 'idle' task, which gets called
 * when no other tasks can run. It can not be killed, and
 * it cannot sleep. The 'state' information in task[0] is
 * never used.  */
asmlinkage void schedule(void)
{
  struct schedule_data * sched_data;
  struct task_struct * prev, * next;
  int this_cpu;

  prev = current;
  /* move an exhausted RR process to be last.. */
  prev->need_resched = 0;

  if (!prev->counter && prev->policy == SCHED_RR) {
    prev->counter = prev->priority;
    move_last_runqueue(prev);
  }

  switch (prev->state) {
    case TASK_INTERRUPTIBLE:
      if (signal_pending(prev)) {
        prev->state = TASK_RUNNING;
        break;
      }
    default:
```

```
        del_from_runqueue(prev);
      case TASK_RUNNING:
    }

  /* this is the scheduler proper: */
    {
      struct task_struct * p = init_task.next_run;
      int c = -1000;

      while (p != &init_task) {
        if (can_schedule(p)) {
          int weight = goodness(p, prev, this_cpu);
          if (weight > c)
            c = weight, next = p;
        }
        p = p->next_run;
      }

      /* Do we need to re-calculate counters? */
      if (!c) {
        struct task_struct *p;
        for_each_task(p)
          p->counter = (p->counter >> 1) + p->priority;
      }
    }

    if (prev != next) {
      /* Se incrementa el número de cambios de contexto (contabilidad)*/
      kstat.context_swtch++;
      switch_to(prev,next);

    }
    return;
  }
```

MACRO SWITCH_TO

El cambio de contexto es el mecanismo que permite que un proceso ceda el control del procesador a otro proceso, retomando más tarde su ejecución por donde la dejó. Esto permite disponer de más procesos que procesadores.

Los microprocesadores de la arquitectura i386 disponen de soporte para multitarea: gestión de estado del procesador por proceso, gestión de mapas de memoria por proceso y mecanismo de cambio de contexto.

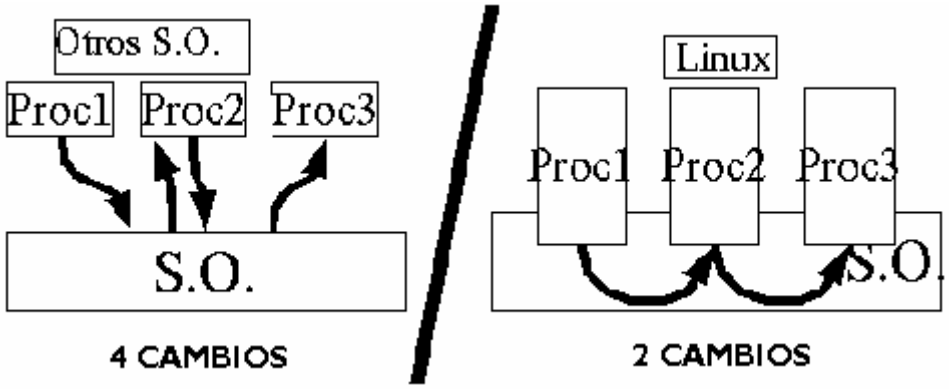
En la arquitectura i386 se denomina *task* a cada uno de los hilos de ejecución. Cada una de estas *tasks* será un proceso de Linux.

Desde el punto de vista del procesador, el estado de una tarea es toda la información que el procesador necesita para poder ejecutarla. Cuando la tarea está en ejecución el estado de la tarea se compone de todos los registros del procesador.

El i386 dispone de instrucciones para guardar y reponer el estado del procesador (estado de la tarea en ejecución) desde memoria principal. Este cambio de estado se denomina *cambio de contexto*.

En muchos S.O. actuales el núcleo es un proceso; y por tanto la forma de acceder al S.O. es mediante un cambio de contexto. Sin embargo, **el núcleo de Linux NO es un proceso.**

En Linux el cambio de contexto se realiza bajo demanda. El núcleo (el proceso en ejecución, dentro del núcleo) ejecuta una instrucción para cambiar a otra tarea (que también estará en el núcleo, en el punto en que se quedó al ceder la CPU). Los procesos Linux son corrutinas que se ceden el procesador de forma explícita.



El i386 dispone de un segmento llamado *Segmento de estado de tarea* (***Task State Segment,*** TSS) que es un segmento dedicado exclusivamente a contener el estado de una tarea cuando no está en ejecución. Un "segmento" en el i386 es un área de memoria (base y tamaño) referenciada por un "descriptor de segmento".

Linux guarda la TSS de cada proceso en el campo *struct thread_struct tss* de la tabla de procesos *task_struct*

En el i386 se pueden invocar hasta 4 mecanismos distintos para desencadenar el cambio de tarea "automáticamente". Sin embargo, por cuestiones de flexibilidad y seguridad, a partir de la versión 2.2 de Linux, el cambio de contexto está codificado paso a paso. En los comentarios del código se aduce que el tiempo requerido por ambas alternativas es en estos momentos prácticamente el mismo y sólo en el caso "manual" es posible optimizarlo en el futuro.

Linux almacena una parte del contexto del proceso en la TSS (que queda almacenada en un campo de la tabla de procesos) y otra parte en la pila del núcleo

Como vimos anteriormente, desde la rutina *schedule* se llama a la macro que lleva a cabo en su interior el cambio de contexto (cuando retorna ya no se está ejecutando prev, sino next).

Veremos muy someramente el código de la macro *switch_to* y el de la función que ejecuta.

Las instrucciones ensamblador contenidas en la macro, tras preservar algunos registros, cambian el puntero de pila de prev a next. Este es uno de los puntos clave del cambio de contexto, ya que aquí es donde la pila de prev, asociada a su entrada en la tabla de procesos, deja de ser la pila del procesador, pasando a ocupar este papel la pila de next. De esta forma, cuando se vuelva a código de usuario, se restaurarán los registros que guardó SAVE_ALL en la pila cuando next saltó a código de núcleo. Finalmente se llama a la función **C** *__switch_to*. Las últimas instrucciones antes del salto se aseguran de salvar en la pila la dirección de retorno para que el control vuelva a la instrucción siguiente al salto (que no es una llamada a subprograma, call sino un salto a la etiqueta 1 situada a continuación del jmp). Una vez dentro de la función *__switch_to*, la función se empieza por guardar los registros de coma flotante si es necesario. A continuación, cargamos en TR (registro que apunta a un slot la GDT. A su vez ese slot apunta a un lugar especifico dentro del TSS del nuevo proceso) el valor de este registro guardado en la TSS de next. Después, guardamos en su TSS los registros fs y gs, de prev y poco después recargaremos los de next. Los registros de segmento gs y fs se usan en la comprobación de acceso a direcciones de memoria desde el núcleo y por eso deben guardarse (dependen del proceso en ejecución).

Después, cargamos las tablas de descriptores locales (LDT) y de páginas (registro CR3) de next sólo si son diferentes que las de prev (pueden no serlo, por ejemplo, si prev y next son hilos del mismo proceso).

Finalmente, si next estaba siendo depurado, cargamos en la CPU los registros de depuración.

En definitiva, el proceso en modo usuario ha guardado en la pila del núcleo, asociada a su entrada en la tabla de procesos todos sus registros al cambiar a modo núcleo (en SAVE_ALL) y por tanto el cambio de contexto que se realiza en switch_to se limita a actualizar los registros y descriptores usados en el núcleo y relacionados con el proceso en ejecución. Hay que tener en cuenta que el código que continuará sigue siendo del núcleo. Al volver a modo usuario se restaurará el estado del nuevo proceso en ejecución, next.

```
#define switch_to(prev,next) do { \
        unsigned long eax, edx, ecx; \
        asm volatile("pushl %%ebx\n\t" \
        "pushl %%esi\n\t"
        "pushl %%edi\n\t" \
        "pushl %%ebp\n\t" \

        /* Se guarda el puntero de pila en la TSS (del proceso que va a abandonar la CPU) en
        la posición correspondiente puntero de pila (ESP) */
```

```
            "movl %%esp,%0\n\t" /* save ESP */ \
            "movl %5,%%esp\n\t" /* restore ESP */ \

            /* Se guarda la dirección de la etiqueta 1 en la TSS (del proceso que va a abandonar la
            CPU) en la posición correspondiente al contador de programa (EIP) de esta forma el
            proceso cuando recupere la CPU podrá continuar en dicho punto */
            "movl $1f,%1\n\t" /* save EIP */ \

            /* Se pone en la pila el contador de programa del nuevo proceso (nuevo proceso que
            va a tomar la CPU) para luego hacer un pop y comenzar */
            "pushl %6\n\t" /* restore EIP */ \

            /* que guarda y actualiza los registros de segmento y las tablas de paginas */
            "jmp __switch_to\n" \
            "1:\t" \
            "popl %%ebp\n\t" \
            "popl %%edi\n\t" \
            "popl %%esi\n\t" \
            "popl %%ebx" \
    } while (0)
```

FUNCIÓN GOODNESS

Esta función se utiliza para evaluar la prioridad de los procesos candidatos a ser seleccionados para ser ejecutados.

- Si el proceso ha cedido el procesador voluntariamente entonces se devuelve un 0 limpiando el flag SCHED_YIELD del proceso.

- Si se trata de un proceso en tiempo real (no sea SCHED_OTHER) entonces se devuelve 1000 más su prioridad. Con eso se consigue dar más prioridad a los procesos en tiempo real.

- El valor goodness se calcula sobre la variable local weight. En este punto se sabe que no es un proceso en tiempo real. En un princio weight se inicializa al cuanto de tiempo que le queda (counter). De este modo, los procesos con menos tiempo de procesador serán los más afortunados.

- Se añade una pequeña gratificación si se trata del mismo proceso que el anterior (dos hilos del mismo proceso) que tenía el procesador. Esta operación explota la caché evitando tener que volver a cargar la zona de memoria que es común a ambos hilos.

- Se suma la prioridad del proceso favoreciendo a los procesos con mayor prioridad.

```
/* Valores retornados:
```

```
 *         0: Si decidió ceder la CPU voluntariamente
 *        +ve: cuanto de tiempo restante + prioridad +
                              gratificaciones (procesos en tiempo no real)
 *      +1000: 1000 (procesos en tiempo real) + prioridad  */

static inline int goodness(struct task_struct * p,
  struct task_struct * prev, int this_cpu)
{
  int policy = p->policy;
  int weight;

  if (policy & SCHED_YIELD) {
    p->policy = policy & ~SCHED_YIELD;
    return 0;
  }

  /* Realtime process, select the first one on the
   * runqueue (taking priorities within processes into
   * account).  */
  if (policy != SCHED_OTHER)
    return 1000 + p->rt_priority;

  /* Give the process a first-approximation goodness
   * value according to the number of clock-ticks it has
   * left.
   *
   * Don't do any other calculations if the time slice is
   * over..  */
  weight = p->counter;
  if (weight) {


    /* .. and a slight advantage to the current thread */
    if (p->mm == prev->mm)
      weight += 1;
    weight += p->priority;
  }

  return weight;
}
```

Por último veremos una lista de funciones que son usadas en distintas partes del código para llevar a cabo la planificación de procesos pero que no forman parte de las *funciones principales*:

FUNCION MOVE_FIRST_RUNQUEUE:

Esta función coloca el proceso que se pasa como argumento como primero de la cola de procesos en estado de Preparado.

El proceso se extrae de su posición actual y se coloca tras el proceso *init_task.* La función simplemente actualiza los punteros correspondientes.

```
static inline void
move_first_runqueue(struct task_struct * p)
{
  struct task_struct *next = p->next_run;
  struct task_struct *prev = p->prev_run;

  /* remove from list */
  next->prev_run = prev;
  prev->next_run = next;
  /* add back to list */
  p->prev_run = &init_task;
  next = init_task.next_run;
  init_task.next_run = p;
  p->next_run = next;
  next->prev_run = p;
}
```

FUNCION MOVE_LAST_RUNQUEUE:

Esta función es muy simple y lo único que hace es mover el proceso pasado como argumento al final de la cola de ejecución, es decir, se coloca como antecesor en la cola al proceso *init_task* (que se considera como marca de final de la cola).

Esto se hace redireccionando los punteros de *p* y enlazando el anterior y siguiente a *p* para hacer desaparecer a *p* del medio.

```
static inline void move_last_runqueue(
  struct task_struct * p)
{
  struct task_struct *next = p->next_run;
  struct task_struct *prev = p->prev_run;

  /* remove from list */
  next->prev_run = prev;
  prev->next_run = next;
  /* add back to list */
  p->next_run = &init_task;
  prev = init_task.prev_run;
  init_task.prev_run = p;
```

```
    p->prev_run = prev;
    prev->next_run = p;
  }
```

FUNCION ADD_TO_RUNQUEUE

Esta función añade un proceso a la cola de procesos en estado de Preparado. El nuevo proceso se inserta al comienzo de la cola, justamente detrás del procesos “init_task” que indica el final de la cola circular. Para ello, actualiza los punteros “prev_run” y “next_run” correspondientes e incrementa el número de procesos en la cola.

```
/* Careful!
 *
 * This has to add the process to the _beginning_ of the
 * run-queue, not the end. See the comment about "This is
 * subtle" in the scheduler proper..  */
static inline void add_to_runqueue(struct task_struct *p)
{
  struct task_struct *next = init_task.next_run;

  p->prev_run = &init_task;
  init_task.next_run = p;
  p->next_run = next;
  next->prev_run = p;
  nr_running++;
}
```

FUNCION DEL_FROM_RUNQUEUE:

Esta función simplemente elimina un proceso de la cola de procesos en estado de Preparado. Esto se consigue decrementando el tamaño de la cola y actualizando los punteros del antecesor y del sucesor del proceso que se desea eliminar.

```
  static inline void del_from_runqueue(
    struct task_struct * p)
  {
    struct task_struct *next = p->next_run;
    struct task_struct *prev = p->prev_run;

    nr_running--;
```

```
    next->prev_run = prev;
    prev->next_run = next;
    p->next_run = NULL;
    p->prev_run = NULL;
}
```

FUNCION SIGNAL_PENDING:

Esta función lo único que realiza es devolver verdadero (1) cuando el proceso que se pasa como argumento tiene una señal pendiente o falso (0) cuando no la hay.

```
extern inline int signal_pending(struct task_struct *p)
{
    return (p->sigpending != 0);
}
```

FUNCION SYS_SETPRIORITY:

A continuación vamos a analizar paso a paso el código de *sys_setpriority*, está función permite establecer el nivel de prioridad de un proceso.

- Esta función recibe 3 argumentos que son:
  - **who:** un identificador del tipo indicado por "which"
  - **which:** indica el tipo de identificador que representa "who": un único proceso, un grupo de procesos o todos los procesos de usuario.
  - **niceval:** incremento de la prioridad, en el rango –20 a +19. La función trunca la nueva prioridad al rango permitido por el núcleo.

- Se asegura que el valor de "which" sea válido.

- Para evitar valores negativos (-20..+20), desplaza el valor de niceval en 20 (0..40).

- Realiza un bucle a través de todos las tareas del sistema comprobando cuales satisfacen las condiciones "who" y "which" con la función "proc_sel" y para aquellos que las cumplen se modifica sus prioridades si se tienen los permisos correspondientes.

El código de dicha función es el siguiente:

```
asmlinkage int sys_setpriority(int which, int who, int niceval)

{
```

```
    struct task_struct *p;
    unsigned int priority;
    int error;

   if (which > 2 || which < 0)
     return -EINVAL;

   /* normalize: avoid signed division
    (rounding problems) */
   error = ESRCH;
           /* Codigo reescrito por claridad */
      if (niceval < -19)
        priority = 40;
      else
        if (niceval > 19)
          priority = 1;
        else
          priority = 20 – niceval;
            /* Codigo reescrito por claridad */


   for_each_task(p) {
     if (!proc_sel(p, which, who))
       continue;
           /* Se miran los permisos */
       p->priority = priority;
   }
   return -error;
  }
```

FUNCION SYS_GETPRIORITY:

Esta función es análoga a la anterior, sólo que en lugar de modificar la prioridad de los procesos que cumplen las condiciones “who” y “which”, devuelve la mayor prioridad encontrada.

```
 asmlinkage int sys_getpriority(int which, int who)
 {
    struct task_struct *p;
   long max_prio = -ESRCH;

   if (which > 2 || which < 0)
      return -EINVAL;

   for_each_task (p) {
      if (!proc_sel(p, which, who))
         continue;
      if (p->priority > max_prio)
         max_prio = p->priority;
```

```
  }
  /* scale the priority from timeslice to 0..40 */
  if (max_prio > 0)
     max_prio = (max_prio * 20 + DEF_PRIORITY/2) /
              DEF_PRIORITY;
  return max_prio;
}
```

FUNCION SETSCHEDULER:

Esta función permite cambiar la política de planificación y/o parámetros de la misma. A continuación vamos a analizar paso a paso el código de *setscheduler*:

- Los 3 argumentos de esta función son: *pid* (el proceso objetivo, 0 significa proceso actual); *policy*, la nueva política de planificación; y *param* un struct que contiene información adicional (el nuevo valor para *rt_priority*).

- *setscheduler* copia la *struct sched_param* suministrada desde el espacio de usuario. s*truct sched_param* (16204) tiene un sólo campo llamado *sched_priority* que es la *rt_priority* deseado para el proceso objetivo.

- Encuentra el proceso objetivo, usando *find_process_by_pid* (27608), el cual devuelve un puntero a la tarea actual si *pid* es igual a 0, o un puntero al proceso con el PID dado si existe, o NULL si no existe ningún proceso con ese pid.

- Si el parámetro *policy* es negativo, la política de planificación actual es conservada. De lo contrario, ésta se aceptará si es un valor válido.

- Se asegura que la prioridad está dentro del rango.

- La nueva prioridad del proceso en tiempo real es ahora conocida (entre 0 y 99 inclusive). Si *policy* es *SCHED_OTHER* pero la prioridad del nuevo proceso no es 0, el test fallará. El test también falla si *policy* denota uno de los planificadores de tiempo real pero la nueva prioridad de tiempo real es 0 (si no es 0, esta estará entre 1 y 99 que es lo único que podría ser). En otro caso el test tendrá éxito. Esto no es muy legible pero es correcto, mínimo y rápido.

- Se actualizan los campos *policy* y *rt_priority* del proceso y además, si el proceso está en estado de Preparado (*next_run ¡= null*), se coloca el primero de la cola y se indica que necesita ser replanificado.

```
static int setscheduler(pid_t pid, int policy,
    struct sched_param *param)
{
  struct sched_param lp;
  struct task_struct *p;
  int retval;

              /* Se comprueban los parametros */
```

```
  if (copy_from_user(&lp, param,
              sizeof(struct sched_param)))
    goto out_nounlock;

  p = find_process_by_pid(pid);

  if (policy < 0)
    policy = p->policy;
  else {
    retval = -EINVAL;
    if (policy != SCHED_FIFO && policy != SCHED_RR &&
        policy != SCHED_OTHER)
      goto out_unlock;
  }

                /* Se comprueba que el proceso tiene permisos para cambiar su politica de planificación
                */

  retval = 0;
  p->policy = policy;
  p->rt_priority = lp.sched_priority;
  if (p->next_run)
    move_first_runqueue(p);

  current->need_resched = 1;

out_unlock:

out_nounlock:
  return retval;
}
```

FUNCION RESCHEDULE_IDLE:

Esta función comprueba si el proceso pasado por parámetro puede tener más prioridad para ejecutarse que el actual (puede que sí pero no tiene porque). Si es así se indica que habrá que replanificar el acceso al procesador.

```
static inline void reschedule_idle(
  struct task_struct * p)
{

      /* Si se trata de un proceso en tiempo real o tiene más cuanto de tiempo por consumir que
      el actual se indica que hay que replanificar */
  if (p->policy != SCHED_OTHER ||
      p->counter > current->counter + 3) {
    current->need_resched = 1;
    return;
```

```
 }
}
```

FUNCION WAKE_UP_PROCESS:

Esta función despierta a un proceso estableciendo el estado del mismo a RUNNING y si éste no está ya en la cola de ejecución entonces lo añade a la misma y posteriormente se manda a replanificar la CPU. Esta función es invocada cuando se necesita despertar a un proceso porque ha ocurrido un evento (por ejemplo, ha expirado un temporizador).

```
void wake_up_process(struct task_struct * p)
{
  unsigned long flags;
  p->state = TASK_RUNNING;
  if (!p->next_run) {
    add_to_runqueue(p);
    reschedule_idle(p);
  }
}
```

FUNCION UPDATE_PROCESS_TIMES:

Esta función actualiza el contador de ciclos de procesador restantes para un proceso (cuanto de tiempo por consumir).

sys_setpriority sólo afecta al campo *priority* del proceso, es decir, a la prioridad estática. Esos procesos también tienen prioridades dinámicas representadas por el campo *counter* en cual se vió en *schedule* y *goodness*. También vimos como *schedule* rellenaba cada prioridad dinámica de los procesos, basada en su prioridad estática, cuando el planificador ve que el campo *counter* es 0. Pero no hemos visto todavía la otra pieza del puzzle. ¿Cuándo se decrementa *counter*? ¿Cómo hacer esto al alcanzar 0?

Para sistemas Monoprocesador, la respuesta está en *update_process_times* (27382). *update_process_times* se llama como parte de *update_times* (27412), que se activa al ocurrir una interrupción timer. Como resultado, esta es invocada bastante frecuentemente, unas 100 veces por segundo. Con cada invocación, el *counter* del proceso actual se decrementa en cada iteración por el número de ticks que han transcurrido desde la última vez. Usualmente, esto es un tick, pero el kernel puede perder el número de ticks del timer, por ejemplo, si este está ocupado sirviendo una interrupción en ese instante. Cuando el contador cae por debajo de 0, *update_process_times* establece el flag *need_resched*, indicando que este proceso necesita ser replanificado. Finalmente, se actualizan los campos de contabilidad de la entrada correspondiente al proceso en la tabla de procesos (“*update_one_process*”).

Ahora, debido a que la prioridad por defecto de un proceso (hablando en términos de prioridades del kernel, no en valores de niceness del espacio de usuario) es 20, un proceso obtiene por defecto, un cuanto de tiempo de 21 ticks (si 21 ticks, no 20, ya que el proceso no esta marcado para replanificarse hasta que su prioridad dinámica caiga por debajo de 0). Un tick es una centésima parte de un segundo, o 10 milisegundos, por eso el cuanto de tiempo es de 210 ms, cerca de la quinta parte de un segundo, exactamente como está documentado en la línea 16466).

Un proceso frecuentemente no suele usar su cuanto de tiempo completo ya que con frecuencia debe atender la E/S.

```
static void update_process_times(unsigned long ticks,
                                 unsigned long system)
{
  struct task_struct * p = current;
  unsigned long user = ticks - system;
  if (p->pid) {
    p->counter -= ticks;
    if (p->counter < 0) {
      p->counter = 0;
      p->need_resched = 1;
    }
  }
  update_one_process(p, ticks, user, system, 0);
}
```