Planificación de Procesos

2008

Kilian Perdomo Curbelo
Airam Rodríguez Quintana

Introducción. Conceptos básicos

- Proceso y programa. ¿En qué se diferencian?
- Estados de un proceso
- Estructura Task_Struct
- Cambio de contexto
- Planificación de procesos
- Funciones del planificador

Proceso y programa. ¿En qué se diferencian?

• **Programa:** es una colección de instrucciones que el procesador interpreta y ejecuta. Los programas se almacenan de modo permanente en memoria secundaria. Un módulo cargador los mueve a memoria principal para poder ser ejecutados.

• **Proceso:** es un programa en ejecución. Como consecuencia de ello, el sistema operativo le va a asignar recursos como memoria, dispositivos, archivos, cpu, etc. Tiene su propio contador de programa.

• Es el SO el encargado de lanzar el programa y convertirlo en proceso

Estados de un proceso en Linux

167 #define TASK_RUNNING 0 //En ejecución o listo para ejecutarse

168 #define TASK_INTERRUPTIBLE 1 /*Esperando por algún evento: interrupción hardware, liberación de un recurso por el que espera, que llegue una señal, etc... Si esto ocurre pasa a TASK_RUNNING*/

169 #define TASK_UNINTERRUPTIBLE 2 /*Esperando por algún evento. A diferencia de TASK_INTERRUPTIBLE, no puede ser interrumpido por ninguna señal.*/

170 #define TASK_STOPPED 4 /*Proceso detenido tras recibir alguna señal: SIGTSTP, SIGSTOP, SIGTTIN, SIGTTOU. Reanuda su ejecución cuando recibe la seña SIGCOUNT*/

171 #define TASK_TRACED 8 /*El proceso se está depurando mediante la función ptrace()*/

Estados de un proceso en Linux

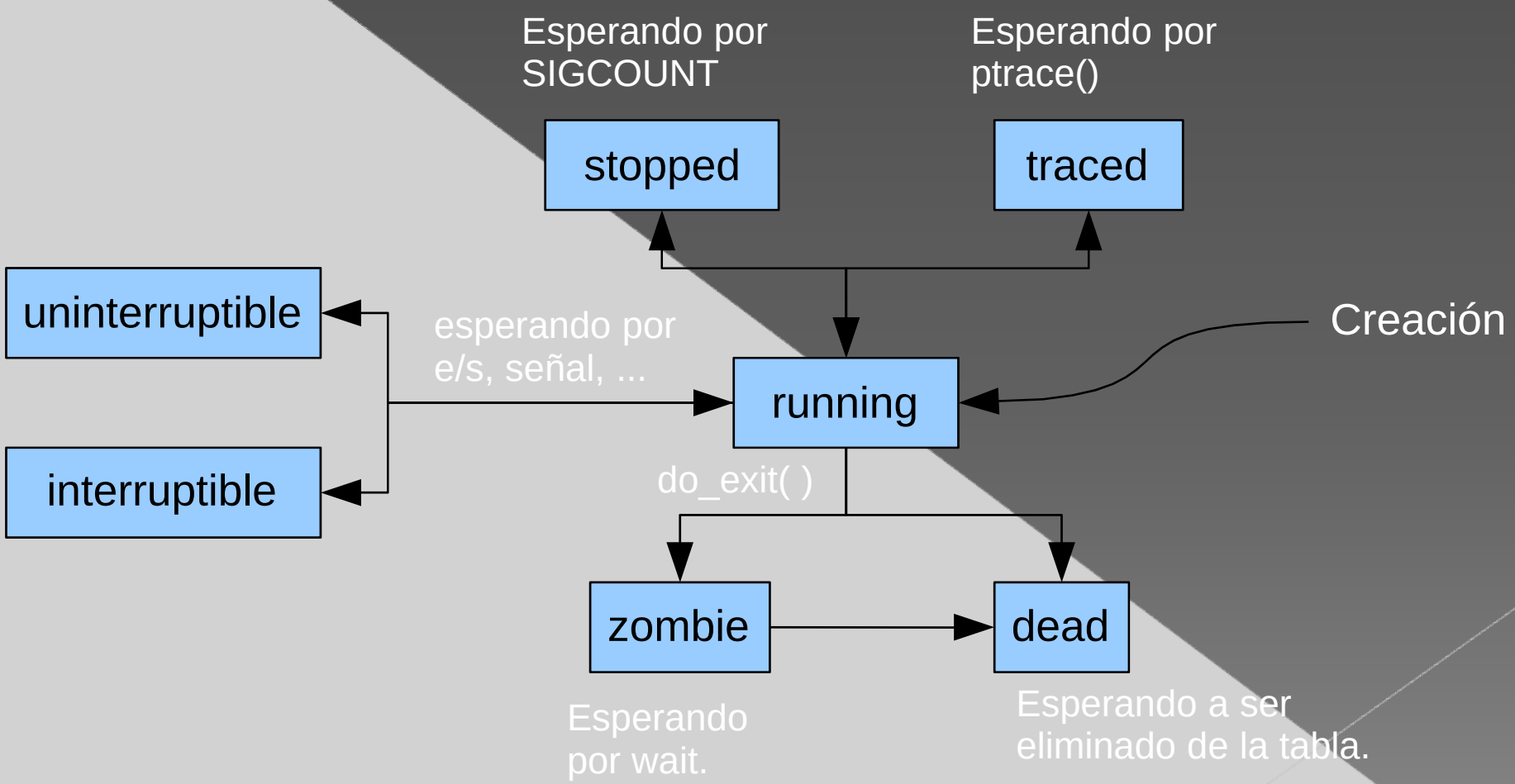
173 #define EXIT_ZOMBIE 16 /*El proceso ha finalizado, pero se mantiene su estructura task_struct hasta que su padre haga un wait()*/

174 #define EXIT_DEAD 32 /*El proceso ha finalizado, y su padre ha hecho wait(). Se libera la estructura del proceso.*/

176 #define TASK_NONINTERACTIVE 64 /*Previene cualquier cambio sobre sleep_avg(calcular prioridad). Asume que todos los procesos en este estado no son interactivos*/

177 #define TASK_DEAD 128 /*El proceso ha sido finalizado mediante do_exit(), poniendo la memoria ocupada por el proceso a NULL. Este es el estado previo a EXIT_ZOMBIE.*/

Diagrama de un proceso en Linux



Estructura task_struct. Descriptor de proceso

- Identificando un proceso
- Listas de procesos
- Listas de procesos en estado Task_Running
- Relaciones entre los procesos
- Campos para expresar la relación de paternidad
- Actualización del estado de un proceso
- Otros campos de la estructura

Introducción a la estructura task_struct

• Cada proceso del sistema en Linux tiene una estructura del tipo task_struct asociada.

• En ella se almacena toda la información relacionada con el proceso: aspectos de planificación, identificadores, relación con otros procesos, memoria utilizada por el proceso, archivos abiertos, etc.

Identificando un proceso

- Cada tarea ha de poder planificarse independientemente, por lo que ha de tener su propio descriptor de proceso (task_struct).
- Cada proceso se puede identificar desde el kernel como un puntero a su **task_struct**.
- Desde el punto de vista del usuario, el identificador del proceso será el PID, que es un campo de la estructura.

```
855   pid_t   pid;     //Identificador del proceso
856   pid_t   tgid;    //Identificador del líder cuando se usan hilos
```

Listas de procesos

Las estructuras task_struct que describen a los procesos, están enlazadas mediante los campos “tasks”, formando una lista circular doblemente encadenada.

La estructura list_head contiene 2 punteros, *next y *prev, que apuntan al siguiente y al anterior, task_struct.

include/linux/list.h

```
28 struct list_head {
29        struct list_head *next, *prev;
30 };
```

include/linux/sched.h

```
553        struct list_head tasks;
```

Listas de procesos en estado Task_Running. Estructura runqueue (I)

Cuando se busca un nuevo proceso para que pase a la CPU, sólo se han de considerar los procesos en estado Task_Running.

La estructura runqueue enlaza los task_struct de todos los procesos en estado TASK_RUNNING exceptuando el proceso idle.

En cuanto a los procesos que están en los demás estados:

- **TASK_STOPPED, EXIT_ZOMBIE, TASK_DEAD y EXIT_DEAD:** No se encuentran enlazados en una lista. No se necesita acceder a ningún proceso que esté en este estado

- **TASK_INTERRUPTIBLE, TASK_UNINTERRUPTIBLE y TASK_NONINTERACTIVE:** cuando el sistema quiere acceder a estos procesos, realiza una búsqueda por PID usando la tabla pid_hash.

Listas de procesos en estado Task_Running. Estructura runqueue (II)

Cada CPU del sistema tiene la suya propia. La macro this_rq( ) proporciona la dirección de la runqueue de la CPU local, mientras que la macro cpu_rq(n) da la dirección de la runqueue de la CPU con índice n.

Cada proceso activo en el sistema pertenece a una cola de procesos ejecutables. Un proceso activo es ejecutado en la cpu a la cual pertenece la runqueue en la que se encuentra el proceso. Aunque los procesos activos también pueden migrar de una runqueue a otra.

```
379  #define cpu_rq(cpu)          (&per_cpu(runqueues, (cpu)))
380 #define this_rq()             (&__get_cpu_var(runqueues))
```

Listas de procesos en estado Task_Running. Estructura runqueue (III)

Las funciones que nos permiten añadir y eliminar los descriptores de las listas de procesos son **enqueue_task y dequeue_task** respectivamente.

```
882 static void enqueue_task(struct rq *rq, struct task_struct *p, int wakeup)
883 {
884         sched_info_queued(p);
885         p->sched_class->enqueue_task(rq, p, wakeup);
886         p->se.on_rq = 1;
887 }
888
889 static void dequeue_task(struct rq *rq, struct task_struct *p, int sleep)
890 {
891         p->sched_class->dequeue_task(rq, p, sleep);
892         p->se.on_rq = 0;
893 }
```

Campos de la estructura runqueue (I)

Tipo	Nombre	Descripción
spinlock_t	lock	Cerrojo para acceder a la lista de procesos en exclusión mutua.
unsigned long	nr_running	Número de procesos ejecutables en la cola de procesos ejecutables.
unsigned long	cpu_load	Carga del procesador en base al promedio del número de procesos existentes en la cola de ejecutables.
unsigned long	nr_switches	Número de cambios de contexto realizados por la CPU.
unsigned long	nr_uninterruptible	Número de procesos que estan en estado TASK_UNINTERRUPTIBLE.
task_t *	curr	Puntero al task_struct del proceso actualmente en ejecución.
task_t *	idle	Un puntero al task_struct del proceso swapper, invocado cuando no hay nada que hacer.

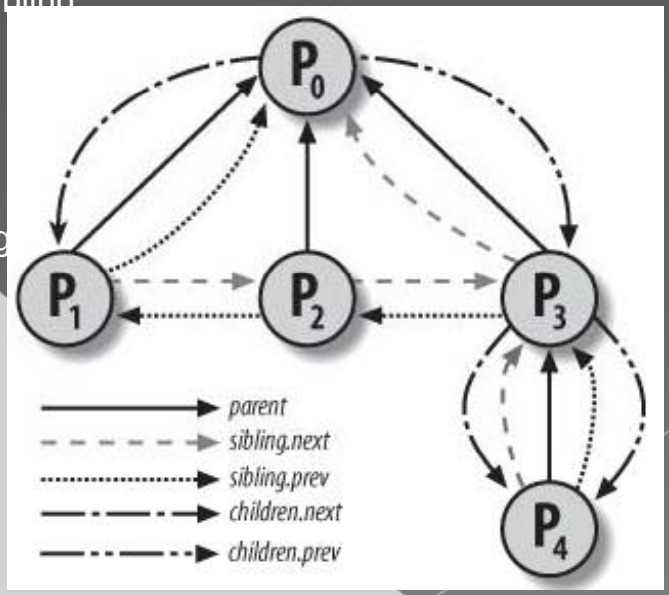
Campos de la estructura runqueue (II)

Tipo	Nombre	Descripción
prio_array_t *	active	Puntero a la lista de procesos activos.
prio_array_t *	expired	Puntero a la lista de procesos que agotaron su rodaja de tiempo (procesos caducados).
prio_array_t [2]	arrays	Dos conjuntos de procesos, los activos y los que agotaron su rodaja de tiempo (procesos caducados)
int	best_expired_prio	La mejor prioridad estática (el valor más bajo) de los procesos que han terminado su rodaja de tiempo (los caducados).
atomic_t	nr_iowait	Número de procesos que estaban en la lista de procesos ejecutables y que actualmente están esperando por una operación de entrada/salida, para completarse.
int	active_balance	Flag que se activa si algún proceso debe migrar de esta cola de ejecución a otra.

Relaciones entre los procesos (I)

Los procesos mantienen una relación padre/hijo (parent/child) con su creador. A su vez los procesos con un padre común tienen relación de hermandad (sibling).

```
1000        /*
1001         * pointers to (original) parent process, youngest child, younger sibling,
1002         * older sibling, respectively.  (p->father can be replaced with
1003         * p->parent->pid)
1004         */
1005        struct task_struct *real_parent; /* real parent process (when being
debugged) */
1006        struct task_struct *parent;     /* parent process */
1007        /*
1008         * children/sibling forms the list of my children plus the
1009         * tasks I'm ptracing.
1010         */
1011        struct list_head children;      /* list of my children */
1012        struct list_head sibling;       /* linkage in my parent's children list */
```



Relaciones entre los procesos (II)

Campo	Descripción
real_parent	Apunta al descriptor de proceso del creador del proceso actual, o al proceso 1 (init) si el padre ya no existe.
parent	También apunta al padre del proceso, al que se le tiene que mandar una señal cuando muera el proceso hijo. Este valor aunque generalmente coincide con el de real_parent, puede diferir si se está usando la llamada al sistema ptrace().
children	Apunta a la cabeza de la lista que contiene a todos los hijos.
sibling	Apunta a la cabeza de la lista de todos sus hermanos.

Actualización del estado de un proceso

Para encontrar fácilmente el proceso en ejecución, hay un puntero (current) que apunta a este proceso.

```
//la macro set_task_state cambia el estado de un proceso determinado
181  #define set_task_state(tsk, state_value)
182          set_mb((tsk)->state, (state_value))


//la macro set_current_state cambia el estado del proceso en ejecución
197  #define set_current_state(state_value)
198          set_mb(current->state, (state_value))
```

Otros campos de la estructura task_struct

Tipo	Nombre	Descripción
unsigned long	state	Estado actual del proceso
int	prio	Prioridad dinámica del proceso.
int	static_prio	Prioridad estática del proceso.
struct list_head	run_list	Punteros al siguiente task_struct y al previo en la cola de procesos ejecutables.
prio_array_t *	Array	Puntero a la estructura prio_array_t que contiene al proceso.
unsigned long	sleep_avg	Promedio de tiempo que el proceso ha estado durmiendo.
unsigned long long	Timestamp	Tiempo de la última inserción del proceso en la cola de ejecución
unsigned long long	last_ran	Tiempo del último cambio de contexto que reemplazó al proceso.
unsigned int long	policy	Política de planificación asociada al proceso. (SCHED_NORMAL, SCHED_RR, SCHED_FIFO or SCHED_BATCH)
unsigned int	time_slice	Ciclos de reloj (ticks) de la rodaja de tiempo que le quedan al proceso.
unsigned int	first_time_slice	Flag a 1 si la rodaja de tiempo nunca expira.

Cambio de contexto (I)

• Dado que todos los procesos comparten el conjunto de registros de cada CPU, habrá que almacenarlos y cargarlos en cada cambio de contexto. Una parte se almacena en el task_struct, y la otra en la pila del kernel.

• El cambio de contexto ocurre en modo kernel, de modo que cuando se vuelve al modo usuario, el contenido de los registros corresponden a los del proceso que ha pasado a ejecutarse.

• Actualmente, el cambio de contexto se ha mejorado aportando una optimización que consiste en postergar el cambio de contexto por más ciclos si se detecta un uso frecuente. De esta manera, el procesador pierde menos tiempo en cambios de contexto, y atiende mejor las operaciones.

Cambio de contexto (II)

- Puntos clave del cambio de contexto: Las instrucciones ensamblador contenidas en la macro, tras preservar algunos registros, cambian el puntero de pila de prev a next.
- 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 __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.

Cambio de contexto (III)

```
15 #define switch_to(prev,next,last) do {                                  \
16         unsigned long esi,edi;                                          \
17         asm volatile("pushfl\n\t"                                       \
18                      "pushl %%ebp\n\t"                                  \
19                      "movl %%esp,%0\n\t"       /* save ESP */           \
20                      "movl %5,%%esp\n\t"       /* restore ESP */        \
21                      "movl $1f,%1\n\t"         /* save EIP */           \
22                      "pushl %6\n\t"            /* restore EIP */        \
23                      "jmp __switch_to\n"                                \
24                      "1:\t"                                             \
25                      "popl %%ebp\n\t"                                   \
26                      "popfl"                                            \
27                      :"=m" (prev->thread.esp),"=m" (prev->thread.eip),  \
28                       "=a" (last),"=S" (esi),"=D" (edi)                 \
29                      :"m" (next->thread.esp),"m" (next->thread.eip),    \
30                       "2" (prev), "d" (next));                          \
31 } while (0)
```

Cambio de contexto (IV)

```
634 struct task_struct fastcall * __switch_to(struct task_struct *prev_p, struct task_struct *next_p)
635 {
636         struct thread_struct *prev = &prev_p->thread,
637                                  *next = &next_p->thread;
638         int cpu = smp_processor_id();
639         struct tss_struct *tss = &per_cpu(init_tss, cpu);
640
641
642
643         __unlazy_fpu(prev_p);
646
647         if (next_p->fpu_counter > 5)
648                 prefetch(&next->i387.fxsave);
650         /*
651          * Reload esp0. */
653         load_esp0(tss, next);
```

Cambio de contexto (V)

```
665     savesegment(fs, prev->fs);
668      /* Load the per-thread Thread-Local Storage descriptor.*/
670     load_TLS(next, cpu);
671
672     /* Restore %fs if needed.  * Glibc normally makes %fs be zero.*/
677     if (unlikely(prev->fs | next->fs))
678             loadsegment(fs, next->fs);
679
680     write_pda(pcurrent, next_p);
683     / * Now maybe handle debug registers and/or IO bitmaps*/
685     if (unlikely((task_thread_info(next_p)->flags & _TIF_WORK_CTXSW)
686         || test_tsk_thread_flag(prev_p, TIF_IO_BITMAP)))
687             __switch_to_xtra(next_p, tss);
688
689     disable_tsc(prev_p, next_p);
691     /* If the task has used fpu the last 5 timeslices, just do a full
692      * restore of the math state immediately to avoid the trap; the
693      * chances of needing FPU soon are obviously high now*/
695     if (next_p->fpu_counter > 5)
696             math_state_restore();
698     return prev_p;}
```

Planificación de procesos

- Política de planificación
- Algoritmo de planificación
- Funciones del planificador

Política de planificación

Consiste en determinar qué proceso y cuándo, ha de pasar a la CPU.

Conseguir objetivos contrapuestos:

- Bajo tiempo de respuesta
- Alta productividad para tareas en segundo plano.
- Evitar la inanición
- Respetar la prioridad de los procesos.
- etc.

La planificación en Linux se basa en el algoritmo round-robin (tiempo compartido) , con la excepción de los procesos que tengan el valor SCHED_FIFO en su campo “policy”, los cuales no pueden ser expulsados hasta que acaben su ejecución.

Política de planificación

- **Prioridad estática**

Cada proceso convencional tiene su propia prioridad estática, de modo que cuanto más alta sea, menor será su prioridad. La prioridad estática se hereda inicialmente del proceso padre. Se usa esencialmente para determinar la duración base de la rodaja de tiempo. Por defecto toma valores entre 100 y 140.

- **Prioridad dinámica**

La política hace uso de prioridades, que son dinámicas. Es decir, se tiene en cuenta la actividad de cada tarea, para ajustar su prioridad periódicamente. De tal modo, que un proceso al que se le ha negado la CPU durante un largo periodo, será recompensado por un aumento de su prioridad; y viceversa. Es la que realmente usa el planificador para elegir el siguiente proceso a ejecutarse en la CPU.

Política de planificación

Se utilizan heurísticas para clasificar los procesos en:

Interactivos (SCHED_RR)

Interactúan con el usuario

-> Mantener bajo el tiempo de respuesta.

En lotes (SCHED_BATCH)

Se encargan de ejecutar tareas en segundo plano

-> Son penalizados por el planificador.

En tiempo real (SCHED_FIFO)

Tareas con tiempo de ejecución acotados.

-> No ceden la CPU a otros procesos

Política de planificación

SCHED_NORMAL: La convencional, de tipo “tiempo compartido”.

SCHED_FIFO: Usada para procesos en tiempo real.

SCHED_RR: Se trata de un round-robin en tiempo real. Reparto equitativo de la CPU entre los procesos con la misma prioridad.

SCHED_BATCH: Consiste en aplicar un bonus o penalización en el rango de -5 a +5 a la prioridad dinámica del proceso. Si la tarea es más interactiva se aplicará un valor negativo, si es consumidora de CPU se aplicará un valor positivo.

Procesos activos y caducados

Los procesos con alta prioridad no deben monopolizar la CPU.

Los procesos activos que terminan su quantum de tiempo pasan a la lista de procesos expirados.

Procesos activos:

Son aquellos que no han agotado su cuanto de tiempo.

-> pueden ejecutarse.

Procesos caducados (expired):

Estos procesos ya han agotado su cuanto de tiempo.

-> no podrán volver a ejecutarse hasta que caduquen todos

Funciones del planificador

Funciones más relevantes:

scheduler_tick( ) → Mantiene contador del cuanto actualizado.

try_to_wake_up( ) → Despierta a un proceso bloqueado.

recalc_task_prio( ) → Actualiza la prioridad dinámica.

schedule( ) → Selecciona el siguiente proceso a ejecutar.

load_balance( ) → Mantiene equilibradas las colas de ejecución (SMP)

try_to_wake_up( ) (I)

Los procesos dormidos se despiertan invocando a esta función, cuando se produce el evento por el que estaban esperando.

```
1398 static int try_to_wake_up(struct task_struct *p, unsigned int state, int sync)
1399 {
1400        int cpu, this_cpu, success = 0;
1401        unsigned long flags;
1402        long old_state;
1403        struct rq *rq;
1404 #ifdef CONFIG_SMP
1405        struct sched_domain *sd, *this_sd = NULL;
1406        unsigned long load, this_load;
1407        int new_cpu;
1408 #endif
```

try_to_wake_up( ) (II)

```
// Se deshabilitan las interrupciones locales y se adquiere el cerrojo de la cola de
// ejecución de la CPU en la que se ejecutó el proceso la última vez
// ( p->thread_info->cpu)

1410            rq = task_rq_lock(p, &flags);

// Se comprueba que el estado esté incluído en la máscara del proceso.

1411            old_state = p->state;
1412            if (!(old_state & state))
1413                    goto out;
1414
// Si p->array no es null, el proceso ya pertenece a una cola de ejecución.
1415            if (p->array)
1416                    goto out_running;
```

try_to_wake_up( ) (III)

```
. . .
     //Se selecciona la CPU en cuya cola de activados se insertará el
proceso
1497          new_cpu = cpu;
. . .
. . .

// Si el proceso estaba en estado TASK_UNINTERRUPTIBLE, se decrementa el
// número de procesos en este estado de la tabla de ejecución del procesador.

1517          if (old_state == TASK_UNINTERRUPTIBLE) {
1518                    rq->nr_uninterruptible--;

//   El   planificador   incrementa   el   sleep   time,   con   el   tiempo
// que el proceso permanece en la cola de ejecución, esto sólo ocurre para los
// estados: TASK_INTERRUPTIBLE y TASK_STOPPED

1523                    p->sleep_type = SLEEP_NONINTERACTIVE;
```

try_to_wake_up( ) (IV)

```
//Se invoca a resched_task para que el proceso despertado pase a ejecución.

1544            if (!sync || cpu != this_cpu) {
1545                    if (TASK_PREEMPTS_CURR(p, rq))
1546                            resched_task(rq->curr);
1547            }
1548            success = 1;

// El nuevo estado del proceso es TASK_RUNNING (ejecutable)
1550 out_running:
1551            p->state = TASK_RUNNING;

// Se libera el cerrojo y se reactivan las interrupciones locales.
1552 out:
1553            task_rq_unlock(rq, &flags);

// Devuelve 1 si se despertó al proceso, y 0 en otro caso.
1555            return success;
```

Schedule( ) (I)

Es la función que implementa el planificador. Sus objetivos son:

- Seleccionar un proceso de la lista de ejecutables.
- Asignarle CPU

Puede ser invocado de dos formas:

- Directa: se produce cuando se lleva a cabo el bloqueo de un proceso.
- Indirecta: se produce cuando el hilo current ha agotado su cuanto de tiempo, cuando se despierta a un proceso y su prioridad es mayor que la del que actualmente está en ejecución,...

Schedule( ) (II)

Por tanto el planificador sustituirá el proceso en ejecución por el que apunte “next”, salvo que no haya ningún proceso con prioridad mayor que el que está en ejecución, en tal caso “next” coincidirá con “current” y no se producirá ningún cambio de contexto.

Schedule( ) (III)

```
3470 asmlinkage void __sched schedule(void)
3471 {
3472        struct task_struct *prev, *next; //Anterior y siguiente proceso
3473        long *switch_count;
3474        struct rq *rq; //Cola de ejecución que se va a tratar
3475        int cpu;
3476
3477 need_resched:
3478        preempt_disable(); //Se desactivan los cambios de contexto
3479        cpu = smp_processor_id(); //Se obtiene el ID de la CPU actual
3480        rq = cpu_rq(cpu); //Se obtiene la cola de esa CPU
3481        rcu_qsctr_inc(cpu);
3482        prev = rq->curr; //El hilo actual pasa a ser el anterior
3483        switch_count = &prev->nivcsw;
3484
3485        release_kernel_lock(prev);
```

Schedule( ) (IV)

/*Antes de decidir qué proceso entra en la CPU, se comprueba que el anterior no fue expulsado en modo expropiativo*/

```
3494        if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
3495                if (unlikely((prev->state & TASK_INTERRUPTIBLE) &&
3496                                unlikely(signal_pending(prev)))) {
3497                        prev->state = TASK_RUNNING;
3498                } else {
3499                        deactivate_task(rq, prev, 1);
3500                }
3501                switch_count = &prev->nvcsw;
3502 }
```

Schedule( ) (V)

```
//La función pick_next_task se encarga de decidir cual es el siguiente proceso
//que pasará a ejecutarse.
3508        next = pick_next_task(rq, prev);
….
3512        if (likely(prev != next)) {
3513                rq->nr_switches++;
3514                rq->curr = next; //El proceso seleccionado se pone como actual
3515                ++*switch_count;
3516
3517                context_switch(rq, prev, next); //Se hace el cambio de contexto
```

load_balance( ) (I)

// Debido a que podemos tener mas de un procesador en una misma maquina y que cada uno de ellos cuenta con su propia lista de procesos, el planificador reparte las cargas entre los procesadores a partir de la función load_balance().

```
2582 static int load_balance(int this_cpu, struct rq *this_rq,
2583            struct sched_domain *sd, enum idle_type idle,
2584            int *balance)
2585 {
2586    int nr_moved, all_pinned = 0, active_balance = 0, sd_idle = 0;
2587    struct sched_group *group;
2588    unsigned long imbalance;
2589    struct rq *busiest;
2590    cpumask_t cpus = CPU_MASK_ALL;
2591    unsigned long flags;
```

load_balance( ) (II)

```
2599    if (idle != NOT_IDLE && sd->flags & SD_SHARE_CPUPOWER &&
2600      !test_sd_parent(sd, SD_POWERSAVINGS_BALANCE))
2601      sd_idle = 1;
2603      schedstat_inc(sd, lb_cnt[idle]);

// Se analiza la carga de los grupos dentro del dominio de la CPU;
// devolviendo el grupo con mayor carga. Salvo que no sea necesario
// hacer un reparto de carga, en cuyo caso devuelve NULL.

2605 redo:
2606 group = find_busiest_group(sd, this_cpu, &imbalance, idle,
             &sd_idle, &cpus, balance);
2609         if (*balance == 0)
2610         goto out_balanced;
2612         if (!group) {
2613                 schedstat_inc(sd, lb_nobusyg[idle]);
2614                 goto out_balanced;
2615         }
```

load_balance( ) (III)

/ / Se busca la CPU con mayor carga del grupo.

```
2617        busiest = find_busiest_queue(group, idle, imbalance, &cpus);
2618        if (!busiest) {
2619                schedstat_inc(sd, lb_nobusyq[idle]);
2620                goto out_balanced;
2621          }
2622
2623          BUG_ON(busiest == this_rq);
2624
2625          schedstat_add(sd, lb_imbalance[idle], imbalance);
2626
2627          nr_moved = 0;
```

load_balance( ) (IV)

```
2674      spin_unlock_irqrestore(&busiest->lock, flags);
2675      if (active_balance)
```

/ / Busca una CPU libre y cuando la encuentra mueve procesos a ella.

```
2676     wake_up_process(busiest->migration_thread);
2678     /* We've kicked active balancing, reset the failure counter.*/

2682     sd->nr_balance_failed = sd->cache_nice_tries+1;
2683     }
2684     } else
2685       sd->nr_balance_failed = 0;
```

Pregunta de examen

Explicar las estructuras más importantes para la planificación de procesos en linux, e indicar donde se decide qué proceso debe pasar a ejecutarse.