Funciones utilizadas por el planificador *0(1)* de Linux

En esta nota se muestran algunas funciones utilizadas por el planificador, centrándose en cómo se inserta un proceso en la cola de preparado que le corresponde dada su prioridad. Como se indica en la pg. 125 de la Guía, ejemplos típicos de la invocación perezosa del planificador se dan cuando:

1. Finaliza el quantum del proceso actual. Entonces el manejador de la interrupción de reloj invoca a la función `scheduler_timer()`.
2. Cuando se despierta un proceso y si prioridad es mayor que la del proceso actual; esto lo realiza la función `try_to_wake_up()`.

- **Función `scheduler_tick()`**

En la página 141 de la Guía se describen las funciones que realiza el manejador de la interrupción de reloj. Una de ellas es la de contabilizar el tiempo de CPU consumido por `current`, y si ha agotado su quantum, apropiarlo. Esto lo realiza la función `scheduler_tick()`, que se invoca en cada tick de reloj. Los principales pasos de esta función son:

1. Almacena en el campo `timestamp_last_tick` de la cola de ejecución local el valor del TSC convertido en nanosegundos (este tiempo se obtiene mediante la función `sched_clock()`).
2. Comprueba si el proceso actual es el swapper de la CPU local; si es así, realiza los pasos:
   a) Si la cola de ejecución local incluye otro proceso ejecutables además del swapper, activa el indicador `TIF_NEED_RESCHED` del proceso actual para forzar la replanificación (podría darse este caso por el soporte que se da al hyperthrearing).
   b) Salta al 7 (no es necesario actualizar el contador de la fracción de tiempo del swapper):
3. Comprueba si `current->array` apunta a la lista activa de la cola de ejecución local. Si no es así, el proceso a consumido su tiempo de quantum, pero no ha sido aún reemplazado: activa `SET_NEED_RESCHED` para forzar la replanificación, y salta a paso 7.
4. Decrementa el contador de la fracción de tiempo del proceso actual, y comprueba si el quantum esta agotado. Las operaciones realizadas por al función son diferentes dependiendo de la clase de planificación del proceso, como veremos en (A). Es una sección crítica protegida por `this_rq()->lock`.
5. Invoca a `rebalance_tick()` que se asegura que las colas de ejecución de las diferentes CPUS contienen aproximadamente los mismos procesos ejecutables.

A) Pasos para actualizar la fracción de quantum en procesos convencionales:

1. Decrementa el contador de la fracción de tiempo (`current->time_slice`).
2. Mira el valor del contador. Si se ha consumido el quantum, se realizan las siguientes acciones:
   a) Invoca a `dequeue_task()` para eliminar `current` de la cola de procesos ejecutables `this_rq()->active`.
   b) Invoca a `set_tsk_need_readched()` para activar `TIF_NEED_RESCHED`.
   c) Actualiza la prioridad dinámica de current:

      ```
      current->prio = effective_prio(current);
      ```

      La función `effective_prio()` lee los campos `static_prio` y `sleep_avg` de `current`, y calcula la prioridad dinámica del proceso de acuerdo a la formula (2).
   d) Rellena el tiempo de quantum del proceso:

      ```
      current->time_slice = task_timeslice(current);
      current->first_time_slice = 0;
      ```

   e) Si el campo `expired_timestamp` de la cola de ejecución local es igual a cero, es decir, el conjunto de procesos expirado es nulo, escribe en el campo el valor del tick actual.
   f) Inserta el proceso actual en el conjunto activo o en el expirado:

      ```
      if (!TASK_INTERACTIVE(p) || EXPIRED_STARVING(rq)) {
              enqueue_task(p, rq->expired);
              if (p->static_prio < rq->best_expired_prio)
      ```

```
            rq->best_expired_prio = p->static_prio;
    } else
            enqueue_task(p, rq->active);
```

La macro `TASK_INTERACTIVE` da el valor 1 si el proceso se reconoce como interactivo usando la fórmula (3). La macro `EXPIRED_STARVING` comprueba si el primer proceso expirado de la cola ha esperado más de 1000 ticks veces el número de procesos ejecutables en la cola de ejecutables más uno; si es así, la macro devuelve 1. También devuelve ese valor si la prioridad estática del proceso actual es mayor que la prioridad estática de cualquier proceso expirado.

3. Si no se agotó el quantum, comprueba si la fracción de tiempo restante de `current` es grande:

```
if (TASK_INTERACTIVE(p) && !((task_timeslice(p) -
          p->time_slice) % TIMESLICE_GRANULARITY(p)) &&
          p->time_slice >= TIMESLICE_GRANULARITY(p)) &&
          p->array == rq->active)) {
                    requeue_task(p, rq->active);
                    set_tsk_need_resched(p);
}
```

La macro `TIMESLICE_GRANULARITY` da el producto del número de CPUs del sistema y una constante proporcional al bonus del proceso actual (Tabla 2.14 de la Guía). Básicamente, el tiempo de quantum de los procesos interactivos con alta prioridad estática se parte en varias piezas de tamaño `TIMESLICE_GRANULARITY`, de forma que no monopolicen la CPU.

Nota: La actualización de la fracción de quantum en procesos de tiempo-real se puede ver en las páginas 271-272 del Bovet&Cesati.

- **Función `try_to_wake_up()`**

La función `try_to_wake_up()` despierta a un proceso dormido o detenido (*stopped*) ajustando su estado a `TASK_RUNNING`, y lo inserta en la cola de procesos preparados de la CU local. Por ejemplo, se invoca para despertar a un proceso que esta en una cola de espera, o para reanudar la ejecución de procesos esperando por una señal. Tiene como parámetros: el descriptor del proceso (`p`) a despertar, una máscara de los estados de los procesos que pueden despertarse, y indicador (`sync`) que prohíbe despertar a un proceso para apropiar al proceso actual en la CPU local.

La función realiza las siguientes acciones:

1. Deshabilita las interrupciones locales con `task_rq_lock()` y adquiere un cerrojo a la cola de ejecutables `rq` propiedad de la CPU que ejecutó al proceso la última vez (puede ser diferente de la CPU local). El número lógico de ésta CPU se almacena en `p->thread_info->cpu`.
2. Comprueba si el estado del proceso `p->state` pertenece a la máscara de estados pasada como argumento a la función; si no es el caso, salta al paso 9 para terminar.
3. Si el campo `p->array` no es `NULL`, el proceso ya pertenece a una cola de ejecutables, por lo que salta a paso 8.
4. En un multiprocesador, comprobamos si debemos migrarlo de la última cola donde se ejecutó.
5. Si el proceso esta en el estado `TASK_UNINTERRUPTIBLE`. Decrementa el campo `nr_uninterruptible` de la cola objetivo, y activa a -1 el campo `p->activated` del descriptor del proceso (ver función `reaclc_task_prio()`).
6. Invoca a la función `activate_task()`, que a su vez realiza lo siguiente:
   a) Invoca a la función `sched_clock()` para obtener el sello temporal en nanosegundos. Si la CPU objetivo no es la local, se compensa la deriva de las interrupciones del cronómetro local utilizando el sello temporal relativo a las últimas ocurrencias de las interrupciones de las CPUs local y objetivo.
   b) Invoca a `recalc_task_prio()`-que veremos después, pasándole el puntero del descriptor del proceso y el sello temporal calculado en el paso anterior.
   c) Ajusta `p->activated` de acuerdo con los valores del punto 10 de la pág. 127 de la Guía.
   d) Asigna el campo `p->timestamp` con el sello temporal calculado en el paso 6a.
   e) Inserta el descriptor del proceso en el conjunto activo:

```
enqueue_task(p, rq->active);
rq->nr_rinnung++;
```

7. Si la CPU objetivo no es la CPU local, o no se ha activado el indicador `sync`, comprueba si el nuevo proceso ejecutable tiene una prioridad dinámica mayor que el proceso actual de la cola `rq`; si es así, invoca a `resched_task()` para apropiar `rq->curr`. En monoprocesadores, la función solo ejecuta `set_tsk_need_resched()`. En multiprocesadores además comprueba si es necesario invocar una interrupción interprocesador para replanificar en la CPU objetivo.
8. Ajusta el estado `p->state` del proceso a `TASK_RUNNING`.
9. Invoca a `task_rq_unlock()` para desbloquear la cola de ejecutables `rq` y rehabilitar las interrupciones locales.
10. Retorna 1 (si el proceso se ha despertado con éxito) o 0 (si no se ha despertado).

- **La función `recalc_task_prio()`**

Esta función actualiza el tiempo de dormido medio y la prioridad dinámica de un proceso. Recibe como argumentos un descriptor de proceso apuntado por `p`, y un sello temporal `now` calculado por la función `sched_clock()`. La función realiza las siguientes acciones:

1. Almacena en la variable local `sleep_time` el resultado de
   `min(now - p->timestamp, 10^9)`
   Si `p->timestamp` contiene el sello temporal del instante del cambio de proceso en que se puso al proceso a dormir, `sleep_time` almacena los nanosegundos que el proceso ha estado durmiendo desde la última ejecución (o 1 segundo, si el proceso empleó más tiempo).
2. Si `sleep_time` no es mayor que cero, salta al 8 (evita actualizar el tiempo dormido medio).
3. Comprueba si el proceso no es una hebra kernel, si ha sido despertado del estado `TASK_UNINTERRUPTIBLE`, y si ha sido despertado continuamente por encima de una umbral de tiempo dormido. Si se satisfacen estas tres condiciones, la función ajusta `p->sleep_avg` al equivalente de 900 ticks (valor empírico obtenido de restar la duración del quantum base de un proceso estándar del tiempo dormido medio máximo). Entonces, salta al paso 8.
   El umbral de tiempo dormido depende de la prioridad estática del proceso. Algunos valores típicos aparecen en la Tabla 2.13. En resumen, el objetivo de esta regla empírica es asegurarse que los procesos que han dormido mucho tiempo en estado no interrumpible -normalmente, esperado una E/S de disco- tienen un valor medio de dormido predefinido, que es suficientemente alto que permita su servicio rápido, pero no tan grande que provoque inanición de otros procesos.
4. Ejecuta la macro `CURRENT_BONUS` para calcular el valor bonus de l tiempo de dormido medio previo del proceso. Si (10-bonus) es mayor que cero, la función multiplica `sleep_time` por este valor. Dado que `sleep_time` se sumará al tiempo medio dormido del proceso, cuanto menor es el tiempo dormido medio actual, más rápidamente crece.
5. Si el proceso esta en modo `TASK_UNINTERRUPTIBLE` y no es una hebra kernel:
   a) Comprueba si el tiempo medio dormido `p->sleep_avg` es mayor o igual al valor umbral de tiempo dormido. Si lo es, pone a cero la variable local `sleep_time` - así, salta el ajuste de tiempo dormido medio- y salta al paso 6.
   b) Si la suma `sleep_time+p->sleep_avg` es mayor o igual al valor umbral, ajusta `p->sleep_avg` al umbral, y `sleep_time` a cero.

   Para limitar algo el incremento del tiempo de dormido medio del proceso, la función no recompensa mucho a los procesos batch que duermen durante largos periodos.
6. Suma `sleep_time` al tiempo medio dormido del proceso (`p->sleep_avg`).
7. Comprueba si `p->sleep_avg` excede 1000 ticks (en nanosg). Si es así, se trunca a 1000 ticks.
8. Actualiza la prioridad dinámica del proceso:
   `p->prio = effective_prio(p);`