Sistemas operativos de tiempo compartido: implementación

Planteamiento: En sistemas multiprogramados se reparte el tiempo del procesador solapando entradas salidas de un proceso con operaciones de computo de otro para alcanzar un mayor rendimiento del procesador. Pero ¿qué pasa si un proceso no realiza operaciones de entrada/salida? Imaginemos el caso extremo de tener 2 procesos un preparado para ejecutarse y otro en ejecución. Si el proceso en ejecución no libera la CPU bien por un error de programación que le haga entrar en un bucle infinito, el proceso en espera de la CPU nunca se ejecutaría.

Objetivo: Modificar el sistema multiprogramado para que podamos retirar la CPU a un proceso en ejecución en determinados momentos y aunque este no la libere voluntariamente. Esta solución debe funcionar independientemente de lo que este haciendo el programa en ejecución. Es decir, implementar un *sistema de tiempo compartido*.

Solución: Debemos asegurarnos de que el sistema operativo tome en control (aunque los procesos no lo cedan).

Como hemos visto, el SO esta controlado por interrupciones/excepciones(trampas), es decir, cuando se produce una interrupción/excepción se deja lo que se este haciendo (tras salvar el contexto de ejecución) y se transfiere el control al manejador de interrupciones/excepciones, o sea al sistema operativo. Por tanto, desde las RSIs debemos quitar el control al proceso actual.

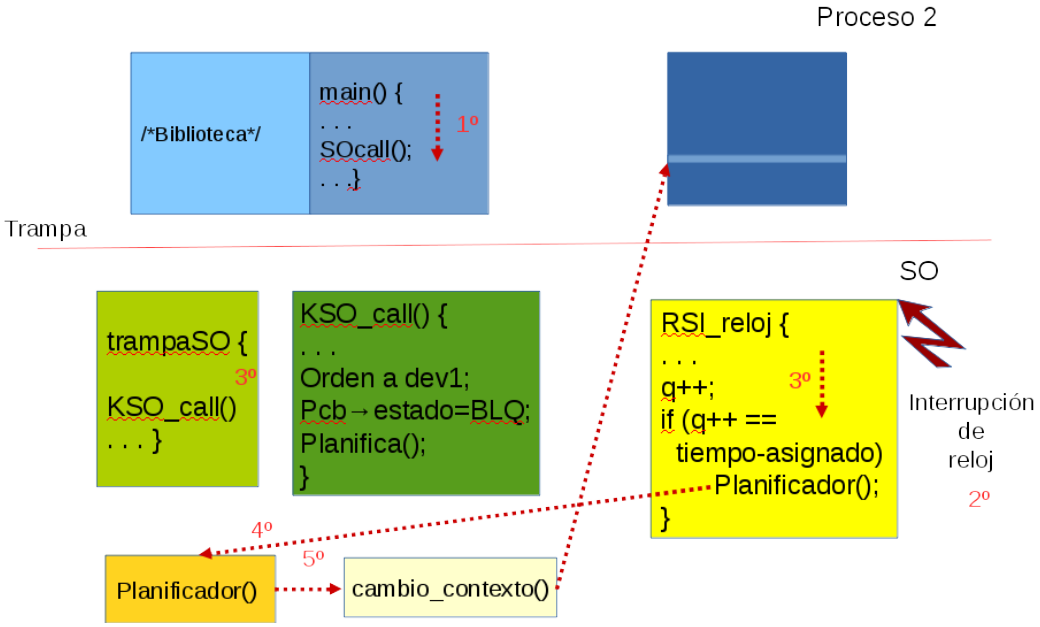
La interrupción que sabemos que se produce de forma regular, independientemente de la actividad de los procesos, es la interrupción de reloj. Por tanto, este es el lugar idóneo para quitar el control al proceso actual de forma periódica.

En el Apartado “2.2 La evolución de los sistemas operativos” epígrafe “Sistemas de tiempo compartido”, pag. 66 (2 párrafo), del Libro W. Stallings, *Sistemas Operativos. Aspectos internos y principios de diseño, 5ª Edición*, Pearson, 2005:

> ria. Un reloj del sistema generaba una interrupción cada 0,2 segundos aproximadamente. En cada interrupción de reloj, el sistema operativo retomaba el control y podía asignar el procesador a otro usuario. Por tanto, a intervalos regulares de tiempo, el usuario actual podría ser desalojado y otro usuario puesto a ejecutar. Para preservar el estado del programa de usuario antiguo, los programas de usuario y los datos se escriben en el disco antes de que se lean los nuevos programas de usuario y nuevos datos. Posteriormente, el código y los datos del programa de usuario antiguo se restauran en memoria principal cuando dicho programa vuelve a ser planificado.

También en Apartado 3.4. “Control de procesos”, epígrafe “Cambio de proceso” pg. 138, epígrafe “Cuándo se realiza el cambio de proceso”, vuelve sobre el tema.

Vamos a retomar el esquema de sistema multiprogramado que veíamos en el Tema 0, y vamos a modificarlo de forma acorde al planteamiento propuesto: Cuando asignamos la CPU a un proceso, se la asignamos durante cierto tiempo, que denominaremos “tiempo-asignado” o *quantum*. Modificaremos la RSI de reloj para que en cada interrupción de reloj (ticks) vaya incrementando una cantidad *q* que se puso a cero al iniciar la ejecución del proceso. Cuando “q < PCB→tiempo-asignado”, la RSI de reloj hace su labor y simplemente le computamos al proceso que se esta actualmente ejecutando el tick de reloj (incrementamos su tiempo de ejecución en una cantidad igual al tiempo que hay entre dos interrupciones de reloj – en Linux, tiene un periodo de 1/HZ segundos, donde HZ se fija en configuración un valor entre 100 y 1000, archivo *include/asm-generic/paran.h*). Ahora bien, cuando el proceso actual consume su tiempo asignado (q == tiempo-asignado) entonces invocamos al planificador, para transferir el control a otro proceso. Ver Figura.



Nota: invocar así al planificador puede presentar problemas de exclusión mutua entre la ejecución de la RSI y la tarea que estuviese ejecutándose cuando se produjo la interrupción, por ello, esta secuencia se resuelve de otra forma ligeramente diferente, como veremos en el Tema 2.

Ejemplo: Veamos como se hace en Linux. Vamos a recurrir a un kernel que tiene el código más "limpio" en este caso. En un kernel 2.4, las RSI de reloj se encuentra en el archivo kernel/timer.c y la función encargada de actualizar el tiempo consumido por el proceso actual es `do_timer()→update_process_time()` que entre otras cosas ejecuta (http://lxr.linux.no/#linux-old+v2.4.31/kernel/timer.c#L595):

```
    /*
 592 * Called from the timer interrupt handler to charge one tick to the current
 593 * process.  user_tick is 1 if the tick is user time, 0 for system.
 594 */
 595void update_process_times(int user_tick)
 596{
 597        struct task_struct *p = current;
 598        int cpu = smp_processor_id(), system = user_tick ^ 1;
 599
 600        update_one_process(p, user_tick, system, cpu);
 601        if (p->pid) {
 602                if (--p->counter <= 0) {
 603                        p->counter = 0;
 604                        /*
 605                         * SCHED_FIFO is priority preemption, so this is
 606                         * not the place to decide whether to reschedule a
 607                         * SCHED_FIFO task or not - Bhavesh Davda
 608                         */
 609                        if (p->policy != SCHED_FIFO) {
 610                                p->need_resched = 1;
 611                        }
```

Comentarios:
- línea 597: la RSI de reloj, selecciona el PCB del proceso actual (`current` es una macro que nos indica la dirección del PCB)
- línea 602: decrementa el contador de tiempo de ejecución asociado al proceso, y se es cero (ha consumido su tiempo), lo pone a cero.
- Línea 610: si el proceso no es de tiempo real (`p->policy != SCHED_FIFO`) entonces planifica, invocando al planificador: `p->need_resched = 1;`