

# Multitarea

El objetivo de la multitarea es que el procesador pueda ejecutar más de una tarea por vez.

## Modelo de loop:

En procesadores de poca capacidad (como por ejemplo los de 8 o 16 bits), se usa un modelo de loop. Una vez terminada la inicialización del sistema, la rutina principal ejecuta un ciclo en el que se ejecutan todas las tareas que se deben realizar. Por ejemplo:



```
int main()  
{  
    init();  
    for (;;)   
    {  
        sense_inputs();  
        process_values();  
        output_control();  
    }  
}
```

Si el ciclo “for” es suficientemente rápido (pocas centésimas de segundo), parece que las tres tareas se ejecutan en paralelo.



Este esquema **no sirve para procesadores modernos**, ya que el procesador estaría activo todo el tiempo innecesariamente **consumiendo mucha energía**, lo que limita el tiempo de uso con batería.

## **Modelo orientado a eventos:**

Una solución más conveniente es la basada en eventos. En este caso, el programa principal se queda durmiendo sin consumir energía. Cuando llega una interrupción, el manejador enciende algún flag y despierta al programa principal, el que ejecuta la tarea correspondiente (por ejemplo: leer el teclado, un carácter del puerto serie, etc).



Para poner el procesador ARM en estado de bajo consumo a partir de ARMv7 existe la instrucción **WFI (Wait For Interrupt)**. En procesadores anteriores ARMv5 y ARMv6 se lograba lo mismo escribiendo en un registro del coprocesador 15.



Existen dos tipos de eventos:

Sincrónicos: ocurrencia determinística y periódica, como los timers.



Asincrónicos: acciones de usuario, como movimiento o click de mouse, pulsar una tecla, etc. O relacionados con la recepción de datos de UART, I2C, Ethernet, etc.



suponiendo que hay un timer y que tengo que atender su interrupcion, debo asegurarme de que cuando me llega una interrupcion asincronica, su handler no debe estar ejecutandose cuando venga otra interrupcion del timer.

En el caso de los eventos sincrónicos, hay que asegurarse que la tarea tarde menos tiempo que el intervalo del timer. Si es más largo, se puede dividir la tarea en dos partes: la urgente o crítica, que se ejecuta dentro del manejador de la interrupción, y el resto, que se ejecuta en el programa principal. Se usan flags para manejar esto. Este esquema también se puede usar para los eventos asincrónicos.

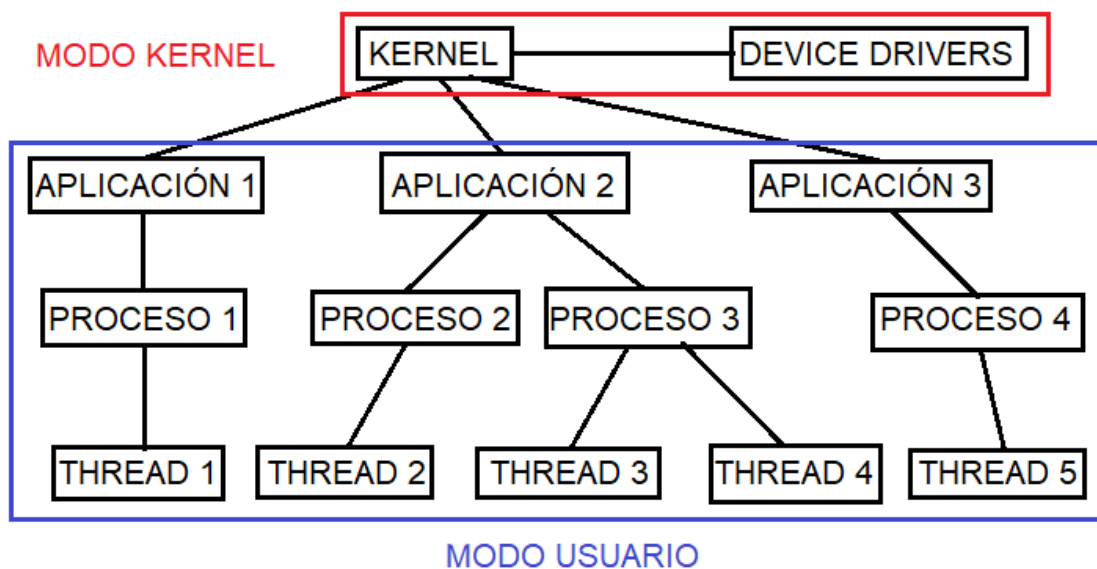
En Linux la parte crítica se denomina Top Half y la otra parte es el Bottom Half.

### Ejemplo:

```
int do_event1() {
    respuesta1_urgente(); event1 = 1;}
int do_event2() {
    respuesta2_urgente(); event2 = 1;}
int main()
{
    init();
    for(;;) {
        if (!event1 && !event2) {
            asm("WFI"); // Esperar nueva int.
        }
        if (event1) {
            lock(); event1 = 0; unlock();
            completar_respuesta1();
        }
        if (event2) {
            lock(); event2 = 0; unlock();
            completar_respuesta2();
        }
    }
}
```

## Modelo de procesos:

Se usa generalmente en sistemas orientados a aplicaciones, donde el sistema puede correr varias aplicaciones simultáneamente.



Un proceso es un programa en ejecución y desde su punto de vista, se supone que tiene todo el sistema a su disposición. Para acceder al hardware y pedir más memoria, debe acceder necesariamente al kernel, que es el que administra los recursos. La interfaz entre los procesos y el kernel se llama API. Un proceso no puede leer ni escribir memoria propia de otro proceso y solo se puede comunicar con otro proceso a través de una API llamada “comunicación entre procesos” (IPC por su sigla en inglés).

Las aplicaciones son abstracciones que corren uno o más programas en uno o más procesos, y el sistema operativo puede tenerlos en cuenta o no.



Los threads son hilos de ejecución que corren los procesos que tienen la particularidad que se ven entre sí y no hay ningún tipo de protección entre dos threads del mismo proceso. Cada proceso tiene al menos un thread.


Cada proceso tiene su propio árbol de tablas de paginación el cual se comparte con todos los threads que corren bajo dicho proceso. El sistema operativo debe escribir esas tablas de manera que un proceso no pueda ver la memoria física de otro proceso.



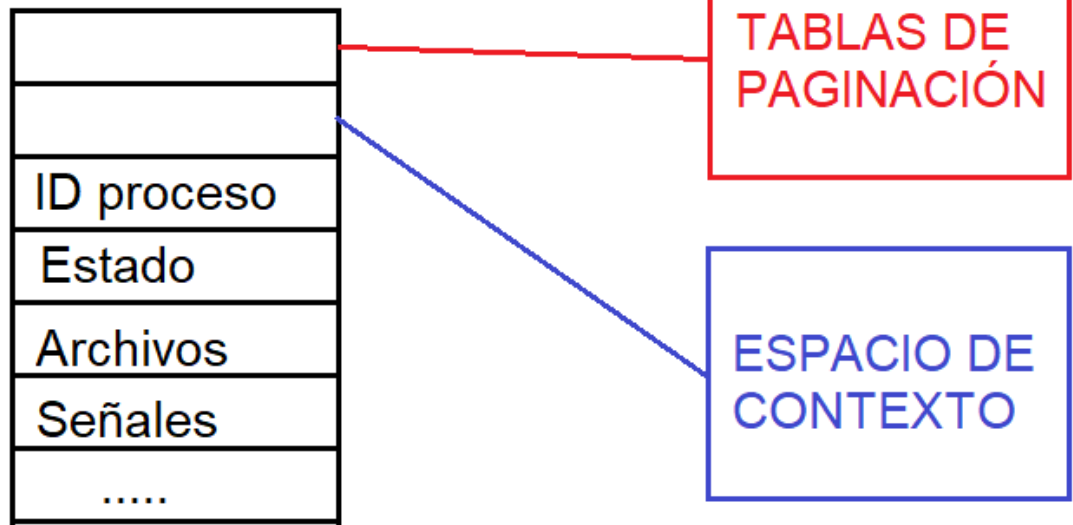
El kernel (núcleo) del sistema operativo y los drivers corren en código privilegiado, lo que les permite acceder a direcciones y hardware inaccesible a las aplicaciones.


Dos partes importantes del kernel son:

- **PCB (Process Control Block):** donde se almacenan los datos que necesita el sistema operativo con respecto a ese proceso en particular, como por ejemplo las tablas

de paginación indicadas arriba, el identificador de proceso, los archivos abiertos y el espacio de contexto (que se explica más abajo). 

### Process Control Block



- **Scheduler:** es el código que determina la siguiente tarea a ejecutar y realiza el cambio de contexto, que ocurre cuando un thread deja de ejecutar para que ejecute un thread diferente. 

Dentro de la clasificación de los sistemas operativos, existe el concepto de sistemas operativos **cooperativos** y **no cooperativos**.

## Sistemas operativos cooperativos (non-preemptive):

En este caso, una tarea corre indefinidamente hasta que ocurra una de las siguientes condiciones:

- La tarea desea detener temporalmente su ejecución usando la llamada al sistema Yield (ceder).
- La tarea llama al kernel para que le entregue datos ingresados por el usuario o de dispositivos de entrada/salida. Si no hay datos, el kernel llama internamente a la función Yield.

La función Yield llama al scheduler para ejecutar otra tarea que esté pendiente.

Se puede observar que, si la tarea nunca llamara a la función Yield, el sistema operativo se colgaría y debería ser reinicializado.



## Sistemas operativos no cooperativos (preemptive):

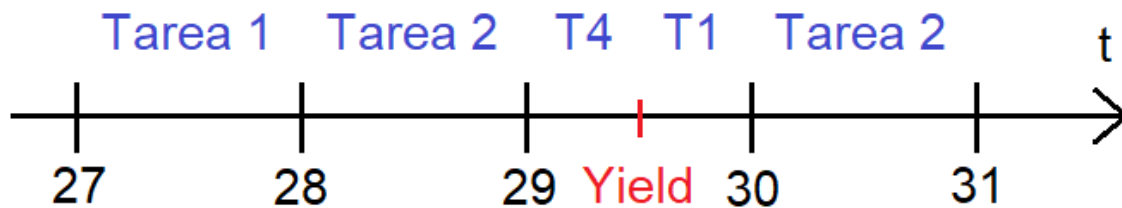
En este caso, que es el que se usa en los sistemas operativos modernos de propósito general, la tarea corre hasta que cede su lugar o bien, ocurre un timeout, lo que ocurra primero. De esta manera la tarea no cuelga el sistema completo. El scheduler en este caso se activa mediante una interrupción de timer.

### **Scheduler y cambio de contexto:**

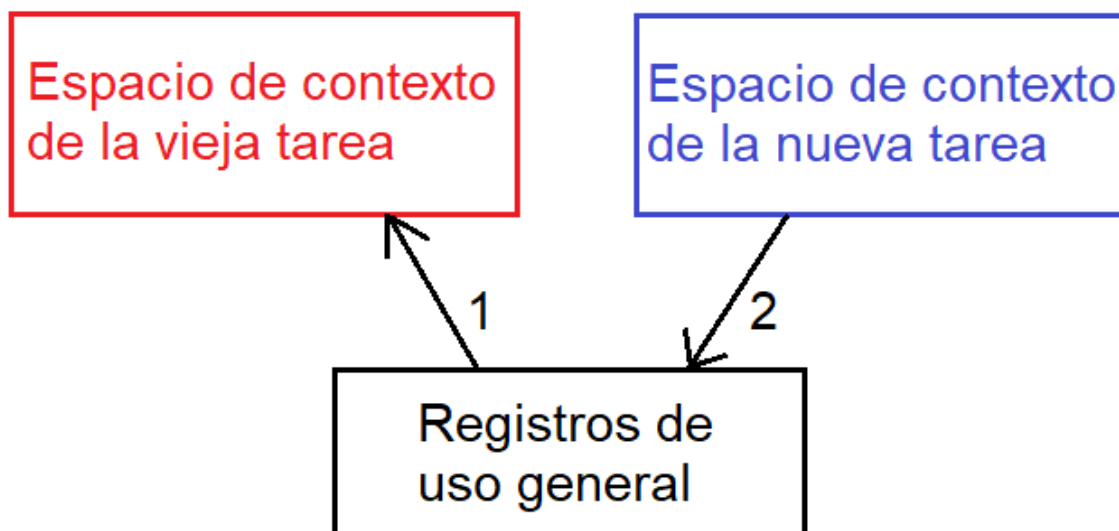
El scheduler se puede dividir en dos partes: el que determina cuál es la próxima tarea por ejecutar y el que hace el cambio de contexto.

Para realizar la determinación, el scheduler determina entre las tareas que estén despiertas cuál es la que corresponde ejecutar, según su prioridad, cuándo se ejecutaron, etc. El método más sencillo es el de Round Robin, que usa una lista enlazada. Cuando es el momento de cambiar de tarea, el scheduler sigue la lista a partir de la vieja tarea buscando alguna tarea despierta. Si la lista se termina, el scheduler vuelve al principio de la lista para seguir buscando.

En el siguiente ejemplo hay 5 tareas, de las cuales 1, 2 y 4 están despiertas y 3 y 5 dormidas (esperando datos).



El cambio de contexto depende del procesador específico. En este momento el sistema operativo salva en el espacio de contexto de la tarea vieja el contenido de los registros de uso general y eventualmente los de SIMD. Luego recupera dichos registros desde el espacio de contexto de la tarea nueva. Finalmente termina el manejador de la interrupción para saltar a la nueva tarea.



Si se usa MMU, hay que modificar el valor de TTBR0 para que apunte a la nueva tabla de primer nivel.

Ejemplo si no se usa SIMD:



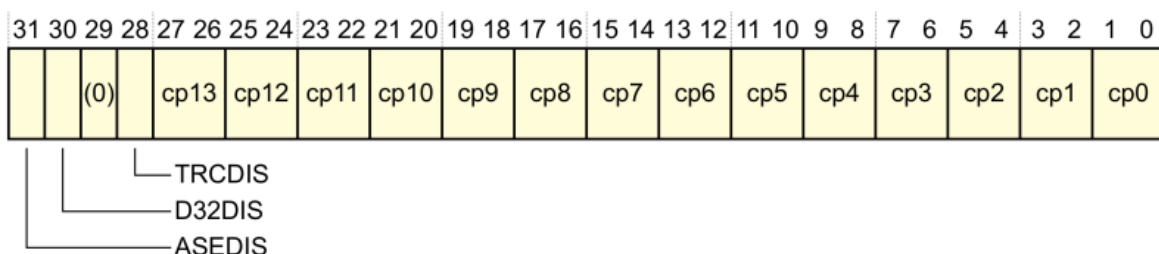
```
SUB LR, LR, #4      ----> Esta linea no iria si fuese un SO cooperativo
PUSH {R0 - R14}
MRS R0, SPSR /* Salvar los flags del viejo modo */
PUSH {R0}
/* Copiar la pila al espacio de contexto de la tarea vieja */
/* Cambiar TTBR0 para que apunte a nueva tabla */
/* Cargar la pila con el espacio de contexto de la tarea nueva
*/
POP {R0} /* Recuperar los flags del viejo modo */
MSR SPSR, R0
POP {R0 - R14}
MOVS PC, LR /* Fin de interrupción */
```

Para que el sistema pueda usar SIMD, los coprocesadores 10 y 11 deben estar habilitados. En caso de no estarlos, cuando el programa ejecute una instrucción de NEON, se produce una excepción de instrucción inválida.

Esto se puede aprovechar de la siguiente manera:

- En el scheduler se verifica si los coprocesadores 10 y 11 están activados. Si están es porque la tarea vieja usó registros SIMD. Entonces el scheduler procede a almacenar todos los registros SIMD en el espacio de contexto de la tarea vieja. Luego deshabilita los coprocesadores 10 y 11.
- En el manejador de la excepción de instrucción inválida se deben habilitar los coprocesadores 10 y 11 y luego restaurar los registros SIMD del espacio de contexto de la tarea que se está ejecutando.

El registro para habilitar y deshabilitar coprocesadores 0 a 13 se denomina CPACR (Coprocesor Access Control Register) y tiene el siguiente contenido:



valor	Acción
00	Acceso Denegado. Genera Undefined Instruction Exception.
01	Acceso en Modo Prilegiado solamente
10	Reservado
11	Acceso en Modo Privilegiado y User

Para accederlo se usan las instrucciones:

MRC p15, 0, <Rt>, c1, c0, 2 /\* Leer CPACR en reg. Rt \*/

MCR p15, 0, <Rt>, c1, c0, 2 /\* Escribir Rt en CPACR \*/

Para nuestros propósitos, los cuatro bits más significativos del registro CPACR deben valer cero.