Diseño e implementación de un núcleo para aplicaciones de tiempo real

Alberto Ballesteros Varela, Bartolomé Palmer Riera

Disseny de Sistemes Operatius, 5º Enginyeria Informàtica

Sumario— **Este documento describe el diseño y la implementación de un núcleo de sistema operativo que da soporte a aplicaciones de tiempo real. El núcleo consiste en un planificador de tareas híbrido que proporciona mecanismos para la ejecución de tareas concurrentes en tiempo real. El núcleo pone a disposición de las tareas primitivas de exclusión mutua; primitivas de sincronización periódicas y aperiódicas; así como una interfaz de entrada y salida de datos adecuada.**

I. Introducción

Un sistema operativo (SO) es un conjunto de programas que se encarga de gestionar los recursos de un ordenador y proporcionar una interfaz con el usuario. Algunos de los sistemas operativos modernos más conocidos son Windows, MacOSX o Linux.

Algunos sistemas operativos dan soporte a sistemas concurrentes, en el sentido que permiten ejecutar varias tareas (programas) de forma virtualmente simultánea. Esto se consigue asignando continuamente la unidad central de proceso (CPU) a cada tarea durante un corto periodo de tiempo.

La concurrencia aporta muchas ventajas, pero también plantea ciertos problemas. Uno de ellos es la sincronización entre tareas. Es necesario proporcionar herramientas que permitan a las tareas sincronizarse y coordinarse en la consecución de un objetivo común.

Otro problema que debe abordarse en sistemas concurrentes se deriva del uso de recursos compartidos. Para cada recurso es necesario garantizar que únicamente un número limitado de tareas (normalmente sólo una) acceda a él simultáneamente. De esta forma, se define el concepto de *región crítica* (RC) como: parte de una tarea, asociada a un recurso, que debe ser realizada sin que otra tarea pueda acceder al mismo. Se dice también que dentro de una región crítica hay *exclusión mutua*, puesto que el acceso de una tarea excluye al resto.

Una de las aplicaciones más importantes de los sistemas concurrentes es el control de *sistemas de tiempo real*. Un *sistema de tiempo real* (STR) es aquel que debe producir resultados correctos cumpliendo plazos de tiempo específicos. Estos sistemas están presentes en escenarios muy diversos: comunicaciones, control de vuelo y satélites, robótica, centrales nucleares, etc. En muchas de estas aplicaciones, un fallo de funcionamiento debido a la violación de un plazo temporal puede tener consecuencias catastróficas.

Un sistema operativo de tiempo real (SOTR) es pues un SO que proporciona mecanismos para que las tareas puedan ejecutarse concurrentemente respetando sus plazos temporales. El respeto de estos plazos es de vital importancia, puesto que en algunos STR basta que una tarea no termine su trabajo dentro de su plazo, para que se considere que el sistema ha fallado.

La entidad más básica de un sistema operativo es el núcleo o *kernel*. El núcleo gestiona la ejecución de las tareas, el uso de la memoria, y provee funciones de entrada/salida de datos a bajo nivel.

Existen diferentes arquitecturas de SO. En esta práctica hemos adoptado una arquitectura de tipo *microkernel*, que consta de un núcleo pequeño optimizado, dónde los añadidos se suman como módulos independientes. En nuestro caso sólo nos interesan las restricciones temporales de las tareas. Por tanto, nos hemos centrado en la implementación de la parte del núcleo que gestiona la ejecución de las tareas: el *planificador*. Adicionalmente, hemos implementado algunas funciones de entrada y salida de datos.

Nuestro SOTR es capaz de dar soporte a la ejecución de un conjunto de tareas concurrentes mediante sencillas llamadas al núcleo. Para demostrarlo, hemos implementado un juego construido a partir de tareas concurrentes que se ejecutan sobre nuestro SOTR.

II. objetivos

Nuestro objetivo es diseñar e implementar un núcleo de SO para aplicaciones de tiempo real. La parte principal es el planificador, que debe encargarse de decidir cuál es la tarea más prioritaria que debe ser ejecutada y de asignar la CPU a esta nueva tarea.

El planificador debe ser híbrido. Es decir, el planificador debe ejecutarse periódicamente y, además, debe ejecutarse para responder a eventos externos aperiódicos, p.e. el pulsado de una tecla.

Un evento externo genera lo que se denomina una *interrupción*. Ésta es una condición que hace que la CPU interrumpa momentáneamente el trabajo que estaba haciendo y ejecute una función conocida como *rutina de servicio a la interrupción* (RSI). El SOTR debe proporcionar al programador una primitiva, que llamaremos *hook*, para que éste pueda insertar una RSI que atienda al pulsado de cualquier tecla.

Además, el núcleo debe proporcionar a las tareas las siguientes primitivas de sincronización y exclusión mutua:

(1) Sincronización periódica mediante la primitiva *delayUntilTime(T)*. Suspende la ejecución de la tarea que la invoca durante *T* unidades de tiempo.

(2) Sincronización aperiódica mediante las primitivas de gestión de flags *waitFlag(flag, mask, condition, option) setFlag(flag, mask)* y *clearFlag(flag, mask)*. Un *flag* es básicamente una variable que contiene una condición. Una tarea que ejecute *waitFlag* se bloqueará hasta que el valor del flag sea el indicado por los parámetros *mascara* y *condición*. Cuando el flag contenga el valor por el cual una

tarea espera, la tarea podrá volver a ejecutarse y cambiará el valor del flag según lo indicado en el parámetro *option*. El valor contenido en un flag puede ser modificado por una tarea o por una RSI mediante las primitivas *setFlag* y *clearFlag*.

(3) Exclusión mutua mediante las primitivas de *semáforos binarios simples wait(S)* y *signal(S)*. Un semáforo binario básicamente es una variable binaria (su valor es *verdadero* o *falso*) que tiene una cola asociada y que protege una región crítica (RC). Una tarea que ejecute *wait(S)* sólo será capaz de entrar en la RC si el semáforo *S* está libre (la variable que indica el valor de *S* está a *falso*). Cuando esto ocurre, *S* pasa a estar ocupado (*verdadero*) por la tarea que hizo el *wait*. Por el contrario, si una tarea ejecuta *wait(S)* y *S* está ocupado, esta tarea se bloquea y espera en la cola del semáforo hasta que éste esté libre. Finalmente, cuando una tarea sale de la RC protegida por *S*, ejecuta *signal(S)*; esto libera a *S* de forma que alguna de las tareas que estaba en la cola de *S* accede a la RC y pasa a estar lista para ejecutarse de nuevo.

(4) Exclusión mutua mediante las primitivas de *semáforos binarios con techo de prioridad inmediato (IPCP) waitIPCP(S)* y *signalIPCP(S)*. Un semáforo IPCP funciona igual que uno simple. La diferencia estriba en que el semáforo IPCP tiene una prioridad asociada. Cuando una tarea consigue ocupar *S*, la tarea adquiere la prioridad de *S* hasta que libera a *S*. Este tipo de semáforo se utiliza para evitar que se produzca un *interbloqueo*: situación en que todas las tareas están bloqueadas esperando en la cola de algún semáforo. Debido a limitaciones de espacio, no entraremos en detalles de cómo se asignan las prioridades a los semáforos IPCP para evitar interbloqueos.

A parte de estas primitivas, el SOTR debe proporcionar a las tareas una función de salida por pantalla que sea adecuada para ellas. No se pueden usar las funciones de salida que proporcionan los lenguajes de programación convencionales, porque no están preparadas para ser usadas en sistemas concurrentes.

Finalmente, se debe desarrollar un juego de tenis simple (al que llamamos *Pong*) que se ha de ejecutar sobre el SOTR. Este juego estará compuesto por varias tareas periódicas y/o aperiódicas que irán interactuando para realizar toda la lógica y presentación, p.e. para mover la bola y las raquetas.

El sistema debe implementarse sobre una plataforma x86/MS-DOS, y deben desarrollarse con el compilador Borland C++ mediante el lenguaje C+

III. DISEÑO

El diseño está orientado a optimizar la gestión de las tareas. Una tarea es un conjunto de instrucciones que se ejecutan en orden secuencial. Cada tarea se encuentra en un determinado *estado*. La Figura 1 representa las transiciones entre estos *estados*, los cuales son:

- *Activo*. La tarea está lista para ser ejecutada.
- *Ejecución*. La tarea se está ejecutando actualmente en la CPU.
- *Suspendida*. La tarea no está lista para ejecutarse durante una determinada cantidad de tiempo.
- *Bloqueada*. La tarea no está lista para ejecutarse porque está a la espera de algún evento aperiódico (flag) o porque está esperando en la cola de un semáforo.

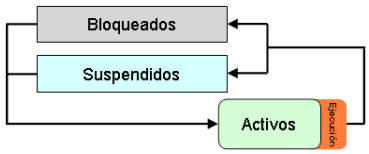

Fig. 1 Estados de una tarea y sus transiciones

Por otra parte, cada tarea tiene reservado un espacio de memoria, llamado pila, que tiene dos utilidades principales. (1) Permitir a la tarea guardar información sobre sus variables locales; (2) permitir a la tarea pasar parámetros y obtener resultados cuando realiza una llamada a función. Una función es un conjunto de instrucciones que realizan un cometido determinado que diferentes tareas podrían necesitar, p.e. varias tareas podrían invocar a una función llamada "calcularSen(2π) ", donde '2π' se pasaría como parámetro.

Además, la pila también permite guardar parte de la información que constituye el *estado de ejecución* de una tarea, cuando ésta es desplazada de la CPU. Es importante no confundir el *estado* de una tarea con su *estado de ejecución*. El *estado de ejecución* de una tarea consiste en el valor que tienen todos y cada uno de los registros de la CPU en un momento determinado de su ejecución, p.e. el valor del registro *program counter* indica cuál es la siguiente instrucción que la tarea va a ejecutar.

El hecho de que la pila permita guardar parte del *estado de ejecución* de su tarea es útil para el planificador. Cuando éste decide que una tarea diferente debe ocupar la CPU, realiza lo que se denomina *intercambio de contexto*. El *intercambio de contexto* consiste en guardar el *estado de ejecución* de la tarea que actualmente se está ejecutando, pero que va a ser desplazada, y restaurar el *estado de ejecución* de la nueva tarea que va a ocupar la CPU. Una vez hecho esto, el planificador indica a la CPU que tiene que seguir ejecutando la nueva tarea a partir del punto donde ésta se interrumpió cuando fue desplazada en el pasado. La Figura 2 muestra un ejemplo de este procedimiento.

Sin embargo, la pila no es suficiente para guardar toda la información necesaria para gestionar una tarea. Por ello, el planificador representa y guarda información adicional sobre una tarea determinada en una estructura de datos dedicada que se denomina *Task Control Block* (TCB).

Como veremos en la siguiente sección, los TCBs se agrupan en diferentes conjuntos, dependiendo del *estado* de las tareas que representan. Por ejemplo, habrá un conjunto que contendrá los TCBs de las tareas activas, otro con los TCBs de las tareas suspendidas, etc.

Un TCB concreto incluye la siguiente información:

- La pila de su tarea.
- Los datos sobre el *estado de ejecución* de su tarea que no pueden guardarse en la pila de ésta. Estos datos son imprescindibles para realizar el *intercambio de contexto*.
- La prioridad de su tarea.

- El instante de tiempo en el que su tarea debe despertarse, después de haber usado la primitiva *delayUntilTime*.
- La condición por la que su tarea está esperando para desbloquearse tras haber usado la primitiva waitFlag.

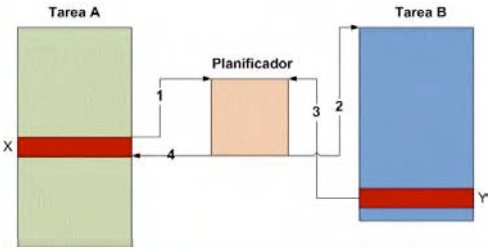

Fig. 2 Ejemplo de cómo dos tareas se alternan en la CPU

IV. IMPLEMENTACIÓN

Como se ha dicho antes, la información necesaria para gestionar una tarea se almacena en su TCB. Los TCBs se agrupan en diferentes conjuntos según el *estado* en el que se encuentren las tareas que representan. Cada conjunto de TCBs se implementa mediante la estructura de datos que más convenga. Pueden verse los conjuntos de TCBs en la Figura 3.

Básicamente hemos implementado dos tipos de estructuras: *lista simple* y *montículo*; si bien hemos utilizado también algunas variantes de la lista simple. Una lista simple es un conjunto de elementos no ordenados y almacenados de forma secuencial. Las inserciones se efectúan por el final y tienen un coste constante. Sin embargo, se necesita recorrer la lista para encontrar un elemento determinado almacenado en ella. Hemos implementado cada lista simple sobre un vector que tiene un tamaño máximo predefinido.

En un montículo los elementos se ordenan mediante una clave proporcionada en el momento de la inserción. Para mantener los elementos ordenados, éstos se organizan en forma de árbol binario. De esta manera, cada inserción tiene un coste logarítmico y las consultas tienen un coste constante. Un problema inherente a la implementación de esta estructura, es que sólo permite leer el elemento que tiene más prioridad.

La tabla I muestra los costes de las diferentes operaciones que se pueden realizar sobre estas estructuras.

TABLA II
COSTES OPERACIONES DE CADA ESTRUCTURA DE DATOS

	Inserción	**Eliminación**	**Consulta**
Lista simple	Constante	Constante	Lineal
Montículo	Logarítmico	Logarítmico	Constante

A continuación se describen los conjuntos de TCBs que se han implementado, así como la estructura de datos que se ha seleccionado para cada uno de ellos (Figura 3).

Es necesario remarcar que sólo *listaTarea* contiene TCBs; el resto de listas almacenan un apuntador (una referencia) a un TCB en concreto. De esta manera, sólo se almacena un TCB por tarea, con lo que se consigue ahorrar mucha memoria, sin pérdida de velocidad.

4) Conjunto de tareas (*listaTarea*): se implementa con una lista indexada por el identificador de la tarea. De esta manera, la lista permanece ordenada y el coste del acceso es lineal.
5) Conjunto de tareas activas (*activos*): está implementado con un montículo, ya que es necesario que estén ordenadas por prioridad.
6) Conjunto de tareas suspendidas (*suspendidos*): implementado con una lista simple, ya que es necesario recorrer todos los elementos.
7) Conjunto de tareas bloqueadas por diferentes semáforos (*listaBloqSem*): está implementado con una lista. Cada elemento contiene un montículo donde están almacenadas las tareas bloqueadas por un semáforo determinado.
8) Finalmente tenemos las tareas bloqueadas a causa de flags (*listaBloqFlag*): es el mismo caso que las tareas suspendidas. Se utiliza una lista simple, ya que es necesario recorrer todos los elementos de esta lista.

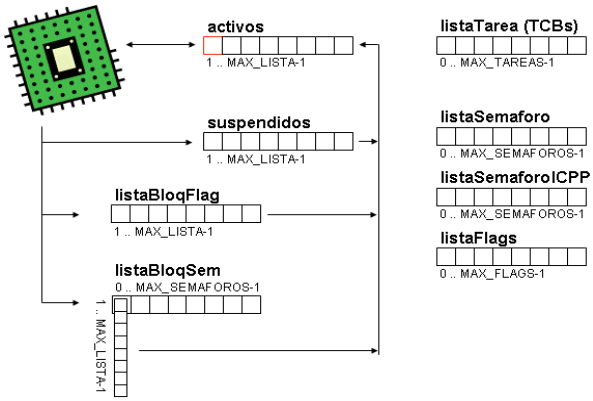

Fig. 3 Representación gráfica de los conjuntos de TCBs

Como se ha dicho anteriormente, el planificador utiliza estas estructuras y la información contenida en los TCBs para gestionar las tareas: para decidir qué tarea debe ejecutarse en cada momento y para realizar el *intercambio de contexto*.

La mayor parte del planificador está implementado dentro de una función, llamada *scheduler()*, que puede invocarse desde distintos puntos; básicamente, desde una tarea que ejecuta una primitiva de sincronización, o desde una rutina de servicio a la interrupción (RSI).

Sin embargo, parte del trabajo que realiza el planificador está implementado en las propias primitivas de sincronización. Es decir, una primitiva de sincronización puede realizar ciertas operaciones sobre los conjuntos de TCBs y, luego, invocar al *scheduler* para que realice el trabajo restante.

Notar que al estar ejecutándose en un entorno concurrente, una primitiva de sincronización tiene que asegurarse de que accede de forma exclusiva a los conjuntos de TCBs. Para ello utiliza una variable global que indica que está accediendo a datos del planificador. En la Figura 4 se ve un ejemplo de cómo se implementaría esta exclusión mutua. Cualquier RSI que quisiese invocar al *scheduler* debería verificar que la variable *planificadorOcupado* está a falso. Por tanto, al poner la variable *planificadorOcupado* a verdadero, la primitiva evita que una RSI pueda invocar al *scheduler* (el cual modificaría los TCBs) mientras ella los está manipulando.

```
primitiva_sincronización()
    {
       planificadorOcupado = verdadero
            acceso exclusivo a los conjuntos de TCBs
            si es necesario, llamar al scheduler
       planificadorOcupado = falso
    }
```

Fig. 4 Esquema de primitiva de sincronización para asegurar exclusión mutua

En cuanto a las rutinas de servicio a la interrupción (RSI), éstas sólo se ejecutan si ocurre cierto evento. Estos eventos están definidos físicamente en la arquitectura del propio ordenador. Cuando ocurre cierto evento se ejecuta su RSI asociada predeterminada.

Sin embargo, el ordenador permite redefinir la RSI que se ejecutará en cada evento. De esta manera, no sólo podemos instalar nuestras propias rutinas (RSIs del núcleo del SOTR), sino que podemos ofrecer al usuario instalar las suyas.

A las RSIs definidas por el usuario las denominamos *hooks*. Nuestro SOTR proporciona funciones para que el usuario pueda instalar y desinstalar *hooks* desde sus programas.

Por otro lado, un caso particular de RSI del núcleo del SOTR es la que controla el tiempo. Funciona usando un cronómetro de la propia arquitectura que tiene diferentes parámetros como el tipo de cuenta o el valor inicial. Variando estos parámetros se consigue que cada cierta cantidad de tiempo se ejecute nuestra RSI, que a su vez llama al *scheduler*. Esta RSI es muy importante; sin ella el SOTR no tendría un conocimiento exacto del tiempo que transcurre ni podría invocar al planificador periódicamente.

Finamente, se ha implementado el juego (*Pong*) que se ejecuta sobre el SOTR. Está compuesto por varias tareas, periódicas y aperiódicas, que irán interactuando para realizar toda la lógica y la presentación. Existe una tarea para cada una de las siguientes funcionalidades:

9) Gestionar el pulsado de cualquier tecla para actualizar las variables que determinan aspectos como: la posición de cada pala, la velocidad de la bola y la condición de finalización del juego.
10) Actualizar la posición de la bola según su dinámica.
11) Comprobar la posición de la bola y actualizar el marcador, si fuera necesario.
12) Refrescar el contenido que se visualiza en la pantalla.

Fig. 5 Instantánea de la pantalla del Pong

V. CONCLUSIONES

El núcleo es la parte más básica de un sistema operativo y, por tanto, la velocidad con que se ejecutan sus funciones es crucial a la hora de implementar servicios sobre él. Así pues, hemos diseñado e implementado el núcleo intentando optimizar su rendimiento. Por ejemplo, hemos puesto un cuidado especial en la elección de los tipos de datos y en la implementación del intercambio de contexto.

Además, nos hemos preocupado de generar un código modular. El sistema operativo está formado por entidades, lo más independientes posible, que cooperan. Esto hace que el sistema operativo sea más fácil de entender y mantener.

Esta práctica es una primera aproximación a lo que podría ser un sistema operativo comercial. Sin embargo, su realización nos ha ayudado a entender la problemática que se esconde bajo el diseño y la implementación de un planificador de tareas de tiempo real. Hemos desarrollado un tipo de sistema que hemos visto muchas veces funcionando, pero del cual no conocíamos los detalles internos. También nos ha dado la posibilidad de mejorar aspectos de una problemática base usando nuestros conocimientos e intuición.

Asignatura impartida por Manuel Alejandro Barranco González.

Alberto Ballesteros Varela es Ingeniero Técnico en Informática de Sistemas y actualmente está cursando Ingeniería Informática. Trabaja en el diseño, implantación y mantenimiento de sistemas de Terminales de Punto de Venta (TPV). *ballesteros.alberto@gmail.com*.

Bartolomé Palmer Riera es Ingeniero Técnico en Informática de Sistemas y actualmente está cursando Ingeniería Informática. Trabajó durante dos años como técnico informático en el *Consorci d'informàtica local* de Mallorca, realizando tareas de consultoría, helpdesk, y técnico de sistemas. Actualmente trabaja a tiempo parcial como consultor informático. *tolopalmer@gmail.com*