Tema

2

2 Introducción a los Sistemas Operativos de Tiempo Real

La creciente complejidad de la Informática Industrial en muchas ocasiones hace necesario el uso de programas complejos que den respuesta a la gran demanda de requisitos. El programador ha de hacer frente a complicados proyectos en los que gran parte del código es crítico. El Sistema Operativo de Tiempo Real es la solución a muchos de los retos que hay que afrontar, ya que simplifica la gestión de tareas, la seguridad, y el acceso a los recursos de la máquina.

2.1 Conceptos básicos

LA Informática Industrial a menudo trata de la programación de pequeños sistemas informáticos en los que no se tienen los recursos habituales de cualquier ordenador de sobremesa, como son una gran cantidad de RAM, dispositivos de almacenamiento masivo, conexión a redes, herramientas de programación avanzadas, cómodos dispositivos de E/S y otros accesorios. A partir de esta base el desarrollador se enfrenta a la problemática de realizar toda una aplicación de automatización, incluyendo elementos de alto y bajo nivel. Frecuentemente la máquina no dispone de más software que el que escriba el propio codificador; en otras ocasiones, dispondremos de soporte de bibliotecas de funciones específicas suministradas por el fabricante de alguno de los dispositivos. Si la aplicación es demasiado compleja, necesitaremos con casi total seguridad la ayuda de otro programa que nos ofrezca funciones de Sistema.

2.1.1 El Sistema Operativo (S.O.)

El *Sistema Operativo* es un conjunto de programas que dan soporte para la ejecución de aplicaciones. Es esencialmente una colección de programas que ofrecen servicios a otros programas:

- **Administra los recursos** de la máquina, relacionados principalmente con los datos y el *hardware*.- Permite compartir el *hardware* entre aplicaciones, al que protege de ser accedido incorrectamente por las mismas. Reparte el tiempo de procesamiento entre las distintas tareas a realizar.
- **Ofrece sistemas de protección**.- Evita que los programas puedan corromperse entre sí, o monopolizar el uso de alguno de los recursos.
- **Abstrae el acceso al *hardware***.- Hace parecer a la máquina algo superior a lo que en realidad es, creando una sensación de acceso a alto nivel. Aísla al programador de saber cómo interactuar directamente con el *hardware*, y muestra una interfaz del mismo más simplificada y cómoda, adaptada a ser usada desde lenguajes de programación. Gestiona y protege los datos, unifica la Entrada/Salida de los distintos dispositivos, y atiende a las interrupciones. Los SS.OO. orientados a gestionar documentos disponen de sistemas de archivos. Los que requieren interactuar con personas cuentan con interfaces gráficos de usuario, con la que se podría decir que el S.O. crea un sistema virtual para interactuar con la máquina al crear dispositivos software funcionales que en realidad no existen como tales.
- **Se comporta como una biblioteca de funciones**.- Desde el punto de vista de las aplicaciones, se percibe al S.O. como un conjunto de funciones que pueden ser llamadas para proporcionar servicios. Los programadores no tienen que preocuparse así de los asuntos internos de gestión del sistema; para conseguir sus propósitos tan sólo han de saber cómo usar las bibliotecas.
- **Aporta herramientas de desarrollo** para poder crear componentes que añadan nueva funcionalidad al sistema. Los SS.OO. especializados para dispositivos pequeños, como los empotrados, suelen contar con unas herramientas de desarrollo para ser usadas desde otros SS.OO. de propósito general.
- **Gestiona tareas**.- Como un recurso más, el S.O. administra el uso compartido de la CPU, asignando tiempo de ejecución a los programas ejecutables que estén cargados.

- **Mejora la portabilidad de las aplicaciones** entre diferentes configuraciones HW.- Al ofrecer una interfaz normalizada y documentada, permite que una misma aplicación pueda funcionar sin cambios de código fuente en otras plataformas que incorporen los mismos servicios de S.O.

De todos los programas que forman el S.O. son clasificados como *núcleo* (*kernel*) los correspondientes a la gestión a más bajo nivel. Tienen acceso directo al hardware o bien desempeñan funciones críticas de administración. Es la parte más interna e importante de cualquier S.O.

2.1.2 Portabilidad

Una característica deseable en cualquier desarrollo informático es la posibilidad que se tenga para reutilizar código, lo que implica presentes y futuras reducciones de costes de desarrollo. La *portabilidad* de aplicaciones es un término que se utiliza para indicar la capacidad que tienen para ser ejecutadas en distintos sistemas informáticos. Estos se caracterizan principalmente por el tipo de hardware que utilizan, con especial énfasis en el procesador principal. Si varios equipos cuentan con una base de hardware común, decimos que se trata de una plataforma específica. Suele ocurrir que cada plataforma utiliza una familia de procesadores concreta, pero no es ese el único factor que las caracteriza: tipos de buses, arquitectura de memoria, subsistemas gráficos y sonoros, etc. Las plataformas más conocidas son: los compatibles PC (**x86**), los específicos que fabrican Apple (**PowerPC**), Sun (**SPARC**), y Silicon Graphics (**MIPS**). Cada plataforma considerada desde el punto de vista del hardware suele llevar asociada otra plataforma software formada principalmente por el S.O., que caracteriza a su vez a las bibliotecas de programación, y aplicaciones de soporte. Algunos ejemplos son: **Windows**, **MacOS**, **Solaris**, e **IRIX**. Existen por otra parte SS.OO. disponibles para diferentes plataformas hardware, es decir SS.OO. multiplataforma. Tal es el caso de **GNU/Linux**, **FreeBSD**, y otros de menor relevancia, cuyo código se ha adaptado para funcionar sobre distintas configuraciones de procesadores, y hardware de soporte.

Quizá la característica más deseada cuando se habla de "aplicación portable" sea la posibilidad de ejecutarla en diferentes plataformas tanto hardware como software. No obstante existen aplicaciones que ni siquiera son totalmente portables dentro de una misma plataforma, tal vez porque hagan uso de alguna característica hardware o software que no sea universalmente utilizada dentro de ese sistema. Esta ha sido una guerra constante patente en la plataforma PC, que todos sufrimos en alguna ocasión.

La portabilidad de aplicaciones compilables puede considerarse a nivel de:

- **Código binario**.- Se puede ejecutar directamente la aplicación sin necesidad de recompilar o ensamblar ninguna de sus partes. A su vez nos encontramos con programas en forma de:
  - **Código máquina.-** El programa es entendido directamente por la CPU.
  - **Código intermedio.**- El programa ha de ser traducido a código máquina desde una representación intermedia similar a la que generan los compiladores para uso interno. La traducción puede ser interactiva (interpretación no a nivel de fuentes), o diferida (compilación en tiempo de instalación o de ejecución). Se necesita un programa llamado *máquina virtual* que dispone todos los recursos necesarios para le ejecución del código.
- **Código fuente**.- El programa se ejecuta directamente desde el código expresado en un lenguaje de alto nivel, o bien se traduce al código máquina del micro de destino. Es

necesario contar además con las bibliotecas de enlace estático que se vallan a utilizar. Nos encontramos con:

- **Código compilado.-** Recompilando la aplicación para el sistema destino, se obtendrá un ejecutable compatible con el mismo.
- **Código interpretado.**- En la plataforma de destino existe un programa intérprete capaz de ejecutarlo sentencia a sentencia.

Para conseguir la compatibilidad a nivel de código fuente es necesario haber desarrollado la aplicación en un lenguaje que exista en las diferentes plataformas en donde se vaya a compilar. Pero no sólo eso, sino que a menudo termina siendo necesario utilizar el mismo tipo de compilador, ya que las aplicaciones con frecuencia dependen de características específicas de determinados compiladores, que no se encuentran en los demás. Por ejemplo, un programa de tamaño medio desarrollado sobre **Visual C++**, es muy posible que no compile con **gcc**, incluso en un mismo sistema. Para que eso no sea un problema, el programador ha de adoptar ciertas precauciones para no confiar en las peculiaridades de un compilador concreto. Tener en cuenta que ni siquiera los estándares **ISO** son universalmente utilizados al completo por los desarrolladores de herramientas de programación. El compilador genera código específico para una familia de procesadores, por lo que no será posible la ejecución directa mediante otra familia distinta.

Si se tratara de programas interpretados, sólo sería necesaria la presencia del intérprete adaptado al sistema de destino. Los lenguajes de guiones (*scripts*) como **Perl** y **JavaScript** permiten realizar programas interpretados multiplataforma. También los correspondientes a un determinado *shell* o intérprete de línea de comandos como **bash**. En el desarrollo de aplicaciones empotradas, el uso de lenguajes interpretados está casi completamente descartado, debido principalmente a su ineficiencia.

Para asegurar compatibilidad, también sería positivo que la aplicación se ejecutase en el S.O. para el que fue desarrollado. Sin embargo si esta no le llegara a pedir servicios directamente, sino que lo hiciera a través de bibliotecas de programación, existiría la posibilidad de disponer de esas mismas bibliotecas portadas a diferentes plataformas ofreciendo la misma interfaz lo que permitiría su ejecución. En el caso de que se utilicen bibliotecas de carga dinámica, es necesario que se disponga igualmente de ellas en la plataforma destino. Ejemplos de bibliotecas multiplataforma son: **OpenGL** dibujo de gráficos 3D, **OpenAL** gestión del sonido, **SDL** funciones multimedia, **FLTK** y **vxWindows** desarrollo de aplicaciones de escritorio, **GNU libC** biblioteca de tiempo de ejecución de C del proyecto GNU, **KDElibs** sistema de escritorio, **Pthreads** para programación concurrente,… Si quisiéramos desarrollar una aplicación para que funcione en distintas plataformas, pero no queremos tener la necesidad de reprogramarla para adaptarla a cada una de ellas, debemos utilizar sólo los elementos comunes a todos los sistemas objetivo, lo que a menudo implica renunciar a algunas características atractivas presentes únicamente en uno de ellos.

La portabilidad a nivel de código máquina entre diferentes plataformas es mucho más complicada de conseguir. La aplicación ha de disponer de las mismas llamadas al sistema lo que implica que necesita de una idéntica interfaz con: el S.O., las bibliotecas dinámicas y programas de soporte, el hardware al que accediera directamente a bajo nivel, y más concretamente, el mismo tipo de CPU. Los microprocesadores modernos incorporan características avanzadas para permitir la coexistencia de varios subsistemas operativos simultáneamente. Por ejemplo **Solaris** incorpora un módulo que permite ejecutar aplicaciones binarias construidas para **GNU/Linux**, casi todos los SS.OO. para PC ofrecen compatibilidad con la ejecución de aplicaciones desarrolladas para **MS-DOS**, y mediante bibliotecas dinámicas también para **Windows** (en menor grado de conformidad). Cuando no es posible usar alguna de estas

técnicas, siempre queda la emulación por software a nivel de aplicación o mediante hardware adicional dedicado. La emulación[1] por software es por lo general es más lenta, y no respeta los tiempos característicos del hardware original.

Existen otros lenguajes como **Lingo**, o **Flash**, que generan un código intermedio que se descodifica en la plataforma de destino. En esta misma línea nos encontramos entornos como **Java** y **.NET** que presentan soluciones de compromiso para permitir la ejecución en diferentes sistemas por medio del uso de lenguajes intermedios basados en pila similares al código máquina, con compilación a código nativo en tiempo de ejecución o de instalación, y un acceso al sistema oculto tras grandes bibliotecas de programación. El uso de programas ejecutados a partir de código intermedio es habitual a nivel de aplicación de usuario aunque también existen opciones interesantes para la programación a bajo nivel como **InteractiveC** o **Java** para 68HC11.

Para la programación que se va a realizar en posteriores explicaciones del curso, se va a utilizar el estándar POSIX, ampliamente aceptado en muchos Sistemas Operativos. Las siglas POSIX (*The Portable Operating System Interface*) hacen referencia a una especificación de bibliotecas de programación, comandos de línea de órdenes, algunos aspectos sobre los controladores de dispositivo, llamadas al sistema, etc. cuyo objetivo es conseguir la uniformidad entre diferentes SS.OO. Ciñéndonos a esta especificación nuestros desarrollos serán más portables.

Ejercicios

1. Enumere factores que intervengan en la portabilidad de aplicaciones.

- Lenguajes de programación.
- Bibliotecas y otros programas de soporte.
- Compiladores (e Intérpretes).
- Sistema Operativo.
- Procesador.
- Subsistemas de hardware.

2. Justifique brevemente la razón por la que la portabilidad puede no ser un factor importante para el software empotrado.

La portabilidad es una cualidad que indica hasta qué punto un software puede ser ejecutado sin problemas en diferentes sistemas informáticos. Esta es una característica muy buscada en todas aquellas aplicaciones horizontales que vayan a correr en muchas máquinas distintas. Sin embargo debido a que el software empotrado sirve para construir aplicaciones verticales, destinadas y especializadas para un fin concreto y exclusivo, la probabilidad de que pueda ser reutilizado en otro sistema distinto es muy baja. Además se realiza una programación de muy bajo nivel, para estar lo más cercano posible al hardware que se utiliza con el objetivo de controlar y maximizar su eficiencia, lo que colabora en la desestimación del desarrollo de software portable.

[1] Se utiliza el concepto de *emulación* porque la interfaz se la ofrecen a otros programas, y no al ser humano.

2.2 Elementos de un sistema informático

Todo sistema informático apto para admitir un S.O.T.R. ha de contar con una base mínima de elementos en común. A continuación se comentan algunos de ellos, que serán vistos en más detalle.

2.2.1 Procesador

El procesador principal es quizá el elemento más característico del hardware de cualquier sistema informático. La elección del resto de componentes electrónicos está fuertemente influenciada por el tipo de procesador utilizado. Los sistemas de control suelen emplear un solo procesador para realizar tanto la orquestación interna del sistema, como el cómputo a realizar. En estos casos se le suele llamar CPU, o unidad central de proceso. En el caso de que existieran más procesadores en el sistema, estarían supeditados a éste. Existe otra opción menos frecuente en STR consistente en el empleo de varios procesadores equivalentes que se reparten la carga computacional, llevando una relación de igual a igual, o existiendo tal vez uno de ellos con funciones adicionales de control sobre los demás. Este es el llamado multiprocesamiento simétrico (SMP) que no vamos a contemplar aquí debido a que suele ser útil casi exclusivamente para agilizar problemas de cómputo u ordenación de cantidades masivas de datos, muy distintos a los habituales programas de control en los STR.

Los procesadores modernos, concebidos para ir dando respuesta a diferentes trabajos según transcurre el tiempo, incorporan sistemas avanzados de protección. Entre estos se incluye principalmente un mecanismo que permite establecer varios modos de funcionamiento. Estos configuran al micro con distintas prioridades, con el objetivo de limitar el acceso a las funciones más críticas. Se suelen contemplar al menos los dos modos de privilegio: usuario y núcleo, aunque en la realidad se podrían soportan más[2].

El modo usuario es el más restringido y se utiliza para ejecutar aplicaciones. El programador tan sólo tiene libre acceso a los registros de propósito general, puntero de pila, contador de programa, etc., y a la consulta del estado de la máquina. Este modo está diseñado para evitar que las aplicaciones se corrompan las unas a las otras ya sea voluntaria o involuntariamente, al no permitir que se trasgredan ciertos límites.

El modo núcleo deja a los programas total acceso a los recursos del procesador y por tanto al resto del hardware. Desde este modo se tiene acceso a los registros de administración y de control de la CPU, que son útiles para la programación de SS.OO. Una rutina ejecutándose en modo núcleo podría hacer algo tan crítico para el sistema como deshabilitar las interrupciones enmascarables, por poner un ejemplo. Desde este modo se puede conmutar fácilmente al modo de usuario, pero sólo se vuelve a modo núcleo al provocar interrupciones. Los micros de bajas prestaciones que no incorporen varios modos de privilegio funcionan como si siempre estuvieran en modo núcleo.

2.2.2 Tareas (Tasks)

Desde los albores de la informática, ha existido la necesidad de expresar de alguna forma el ente dinámico que forman los programas en ejecución. Como estos habían sido diseñados para resolver alguna tarea en concreto, se les asignó el nombre de *tareas*. Una tarea es cualquier programa que se encuentre cargado en memoria desde la que es procesado por la CPU; Es por

[2] En terminología Intel, se habla de *cuatro anillos* de protección presentes en todos sus micros x86 a partir del 80386. En terminología Motorota se habla del *modo usuario* y el *modo supervisor*.

tanto un ente **dinámico** de ejecución o procesamiento de código. A menudo puede llegar a ser una cuestión subjetiva el discernir dónde están los límites que delimitan las diferentes tareas, por eso se suele considerar que una tarea:

- Debe ser lo más independientes posible de las demás.
- Contará con sistemas para intercambiar información con el exterior y con las demás tareas, usando algún convenio de sincronización que evite la corrupción de datos.
- Dispondrá de áreas de memoria propias en donde almacenar su estado particular.

Dependiendo del contexto, una tarea puede ser desde una simple subrutina, hasta una compleja aplicación de control, siempre teniendo en cuenta que tarea hace referencia más bien a la finalidad del programa. Sin embargo, una rutina excesivamente trivial, difícilmente se la consideraría tarea por independiente que fuera.

Las tareas son por tanto programas en ejecución disjuntos, es decir, que no comparten datos libremente con otros programas en ejecución.

2.2.2.1 Multitarea *(Multitasking)*

Quizá los ejemplos de tareas más conocidos sean las aplicaciones de usuario. Se dice que si un sistema informático sólo pudiera mantener simultáneamente un programa de aplicación cargado en memoria, se trata de un sistema *monotarea*. Si permitiera mantener varios, a los que poder asignar la CPU, hablaríamos de un sistema *multitarea*.

Los sistemas que se implementan sobre DSP's son habitualmente monotarea, al estar especializados en ejecutar un solo programa de control. Todos los sistemas **Windows** y **UNIX** ofrecen soporte multitarea. Otros sistemas anteriores como **DOS** no ofrecían soporte para multitarea, aunque se podía implementar con relativa facilidad, ya que dejaban acceso directo al hardware.

En el caso de que la multitarea sea proporcionada por el SS.OO., tiene el objetivo de ejecutar varios trabajos independientes, transmitiendo la sensación de que están siendo ejecutados en paralelo. Se dice que la multitarea consigue *paralelismo de grano gordo*.

Un sistema es de *multitarea cooperativa* si pasado un pequeño intervalo de tiempo en el que se ha estado usando la CPU, cada tarea cede el control al sistema voluntariamente para dar la oportunidad de que se ejecuten las demás tareas. Un gran inconveniente de la multitarea cooperativa es que cualquier detención en una de las tareas, por ejemplo por la ejecución de un bucle muy largo, detendría a su vez a las demás. La multitarea cooperativa se da en sistemas con núcleos no apropiativos. Ejemplos de tales sistemas son **Windows** 3.11 e inferiores, o en aquellos sistemas que asignan tiempos de CPU basándose en una planificación estática, como se verá en el siguiente capítulo.

Por el contrario decimos que un sistema es de *multitarea apropiativa*, si existe un S.O. capaz de apropiarse de la CPU aunque la tarea no se la haya cedido voluntariamente, con la intención de asignarle el procesador a otra tarea, repitiéndose este ciclo periódicamente. El tiempo máximo que se deja la CPU a cada una de las tareas antes de retirársela forzosamente es del orden de décimas de segundos. Se dice que el núcleo de un sistema de este tipo es apropiativo. Este sistema también se califica como de *multitarea con expulsión*, ya que el núcleo es capaz de expulsar forzosamente a cualquier tarea. Ejemplos de tales sistemas son **Windows XP**, **GNU/Linux**, **MacOS X**, y **FreeBSD**.

2.2.3 Interrupciones

Como ya sabemos, una interrupción es un mecanismo por el cual se comunica a un sistema informático que ha ocurrido un evento. La notificación del mismo se realiza mediante el flanco de subida o de baja de algún terminal eléctrico de los que tiene conectados. Las interrupciones son una de las mejores invenciones para solucionar los problemas de Tiempo Real, ya que permiten obtener respuestas rápidas ante los eventuales cambios de estado externos, permitiendo así mismo maximizar el uso de los procesadores mediante rutinas de fondo. Además si el sistema está bien diseñado, su uso es totalmente transparente al resto de las tareas, lo que es necesario para que sean procesadas independientemente.

El proceso que realiza la CPU para atender a una petición de interrupción o *Interrupt Request* (IRQ), se basa en identificar su tipo, buscar la dirección correspondiente de la *Rutina de Atención a Interrupción* (RAI) o *Interrupt Service Routine* (ISR) que será la encargada de procesarla, salvar en la pila el estado de la tarea que estuviera en ejecución, saltar a la ISR, y procesarla, después de lo cual se recuperará el estado de la tarea que fue interrumpida para continuar con lo que estaba haciendo.

En circunstancias delicadas puede ocurrir que no se desee interrumpir a una determinada tarea de tiempo crítico, debido tal vez a que esté controlando a un dispositivo externo que necesite máxima atención durante un intervalo de tiempo continuado, o se esté realizando una configuración interna delicada. Es por tanto necesario poder contar con la posibilidad de deshabilitar temporalmente las interrupciones, aunque implique un retraso consciente del instante en que se comience a atenderlas, si aparecieran durante dicho intervalo. A estas interrupciones se las clasifica como *enmascarables*, lo que significa que mediante una máscara de bits, pueden ser habilitadas y deshabilitadas. Los sistemas que admitan varias fuentes de interrupción hardware usan un sistema de jerarquización para permitir que unas interrupciones tengan más prioridad que otras, a menudo en función de su importancia. En procesadores simples esta jerarquía afecta a la decisión de qué interrupción entrará la primera en el caso de que durante un intervalo se encuentren deshabilitadas, tal vez porque haya alguna en ejecución, y llegara más de una IRQ. En procesadores más avanzados se implementa un mecanismo de expulsión por el que una nueva interrupción puede imponerse sobre otra de prioridad inferior que se encontrara ejecutando, es decir las IRQ prioritarias interrumpen a su vez a las menos prioritarias.

Existen micros de pocas prestaciones que no implementan interrupciones enmascarables. En estos casos las tareas están obligadas a sondear el exterior para detectar los cambios de estado.

La carencia de interrupciones impide la implementación de SS.OO.T.R., pero no impiden el desarrollo de aplicaciones de tiempo real simples al estilo de los autómatas.

2.2.3.1 Latencia de interrupción, Respuesta y Recuperación

(Interrupt Latency, Response and Recovery)

Existen algunas circunstancias que podrían impedir que las interrupciones sean atendidas inmediatamente después de su solicitud. En los STR es necesario conocerlas para saber cuál es el tiempo máximo que tardaría la ISR en actuar.

En todos los procesadores, sobre todo si son de tecnología CISC, existen instrucciones máquina que no pueden ser interrumpidas. Podría ser que se produjera una solicitud de interrupción durante la ejecución de una de estas instrucciones. En el caso más desfavorable, se trataría de la instrucción de mayor duración de todo el juego de instrucciones, por ejemplo la división de enteros con signo.

El tiempo que empleará el sistema en comenzar el ciclo de reconocimiento de interrupción recibe el nombre de *Latencia de Interrupción*. El cálculo de la máxima latencia de interrupción ha de realizarse para el peor de los casos. Está formado por la acumulación de:

- El tiempo que emplea en ejecutarse la instrucción más larga.
- El periodo de tiempo más largo en el que las interrupciones estén deshabilitadas.
- El tiempo que se tarde en ejecutar las interrupciones de mayor prioridad que hubieran podido solicitar atención durante el proceso.

Pasado este tiempo, la CPU se dispondrá a salvar el estado actual y saltar definitivamente a la ISR. Se llama *tiempo de 1ª Respuesta* al empleado desde que se solicitó la interrupción hasta que comienza a ejecutarse la ISR.

El que comience la ejecución de la ISR es tan solo un paso necesario para generar el resultado. En las ocasiones en las que la línea de interrupción sea compartida por varios dispositivos, es necesario que al principio de la ISR se identifique al hardware que la generó. Esto obligará al procesador a consultar por sondeo a cada uno de las potenciales fuentes de interrupción. Llegado aquí el sistema está en condiciones para realizar el procesamiento pertinente. Para que el dispositivo sepa que ha sido atendido, debe ser notificado de alguna forma, normalmente mediante la escritura en alguno de sus registros internos, o por la simple lectura de su estado. Antes de salir de la ISR, es recomendable comprobar si se ha producido alguna otra solicitud de interrupción proveniente de cualquiera de las fuentes que la comparten. El tiempo total durante el cual la ISR ha tenido el control de la CPU se llama *tiempo de Procesamiento* de la ISR.

Por último, se llama *tiempo de Recuperación* de interrupción al que transcurre desde que se hizo la solicitud hasta que se restaura el estado previo, y se reactiva la tarea que fue interrumpida.

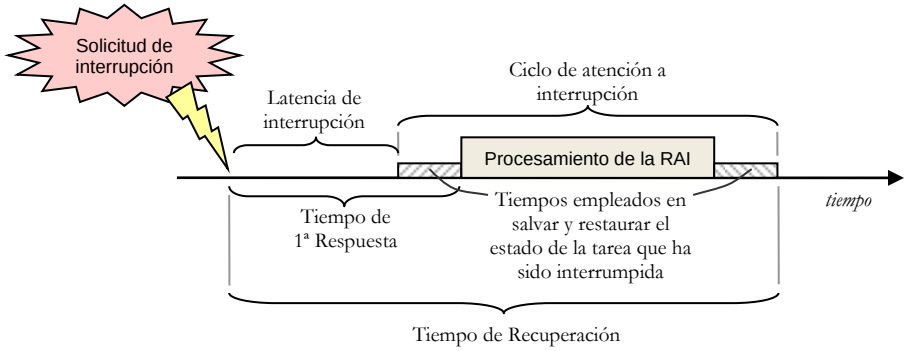


Figura 2-1. Tiempos de latencia, respuesta, y recuperación de interrupción

Todos estos tiempos intervienen en la comparación de rendimiento entre diferentes sistemas. Para minimizarlo deberíamos intentar que:

- El tiempo durante el que las interrupciones se deshabiliten sea lo menor posible.
- El tiempo en procesar una ISR sea igualmente pequeño, para no retener demasiado a otras interrupciones que estén a la espera de que su terminación:
    - La ISR no utilizará bucles cuya cantidad máxima de iteraciones no esté perfectamente delimitada a priori.
    - La ISR deberá realizar un procesamiento de datos simple, no pudiendo entretenerse en efectuar operaciones computacionalmente costosas.

La mayoría de las CPUs vinculan una prioridad a cada línea de entrada de interrupción, ya que se considera que hay eventos más importantes que otros. Las interrupciones anidadas ocurren cuando interrupciones con prioridades altas pueden desalojar a otras de menor prioridad cuyas ISR se estuvieran procesando; tales sistemas se califican también como de expulsión. El anidamiento de interrupciones no existe de forma generalizada en todos los procesadores. Tal es el caso de la mayoría de los de 8 bits, en los que normalmente una ISR normalmente no podrá ser desalojada por otra de mayor prioridad mientras que tenga el control de la CPU, es decir no permiten la expulsión.

De forma simplificada el tiempo de primera respuesta de interrupción para un sistema con expulsión se calcularía con la siguiente fórmula:

T_{Ins} = Instrucción atómica más larga del bucle de fondo.

T_{Int} = Tiempo más largo con las interrupciones deshabilitadas en el bucle de fondo.

C_P = Ciclo de interrupción de las interrupciones de prioridad mayor o igual.

T_R = Tiempo en introducir los registros en pila y dar el salto a la ISR.

$$T_{\text{Respuesta con expulsión}} = máx\{T_{Ins}, T_{Int}\} + \sum C_P + T_R$$

Para el cálculo del tiempo de 1ª respuesta de una interrupción en los sistemas no anidados hay que contemplar la posibilidad de que exista algún ciclo de atención de una interrupción de prioridad menor o igual que haya entrado justo antes de que apareciera esta interrupción:

C_M = Ciclo de interrupción de las interrupciones de prioridad menor o igual.

$$T_{\text{Respuesta sin expulsión}} = máx\{máx\{T_{Ins}, T_{Int}\} + \sum C_P + T_R, C_M\}$$

En los dos casos se ha contado con las siguientes suposiciones:

- El procesador siempre realiza un ciclo de interrupción completo para atender a cualquier interrupción aunque estén seguidas. Si no fuera así, se tendrían que descontar los tiempos que se tarda en salvar y/o recuperar los registros de la pila en las interrupciones que se procesen consecutivamente.
- Los periodos de repetición en los que aparecen nuevas interrupciones son superiores al mayor Tiempo de Recuperación posible.

Lo más habitual es que en los sistemas informáticos grandes contemos con SS.OO. realizados por terceras personas, lo que implica que no podremos escribir directamente el código de las ISR como hacemos con los pequeños microcontroladores. En estos casos lo que se hace es retransmitir el evento a otras tareas que lo están esperando. Para ello desde la ISR se pone un dato en memoria, que más tarde será recibido por la rutina de procesamiento. Esto provoca un retraso en la respuesta muy superior al que se daría si se hubiera realizado el procesamiento correspondiente desde dentro de la propia ISR.

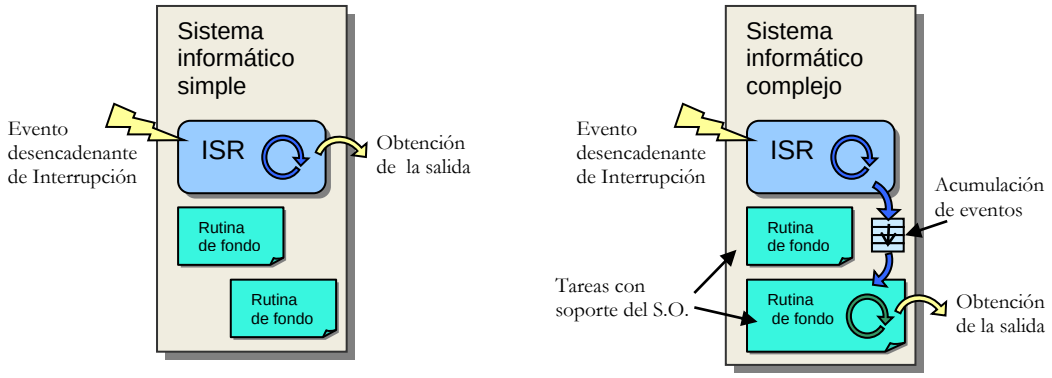


Figura 2-2. Formas de obtener la salida en reacción a un evento.

En la **Figura 2-2** se muestra la diferencia entre ambos tipos de sistemas. En el caso del sistema complejo, es una *rutina de fondo*[3] la encargada de realizar el procesamiento que desencadena el evento. La ISR ha de pasar la información depositándola en alguna parte de la memoria, siendo habitual hacerlo mediante un buffer FIFO software, lo que evitaría la pérdida de datos en el caso de que no hubiera tiempo suficiente para procesarlos durante un determinado número de interrupciones consecutivas. En un sistema informático simple la demora que transcurre hasta que se produce la salida ante un evento de entrada, es del orden de decenas de microsegundos, mientras que en los sistemas más complejos este tiempo puede ser de hasta centenas de microsegundos. Si se tratara de un SOTR ese tiempo puede reducirse hasta unos pocos microsegundos asegurando también un tiempo máximo de respuesta.

Ejercicios

3. **Se tiene un sistema informático basado en microprocesador clásico CISC en el que la instrucción más larga que se ejecuta en el bucle principal de la rutina de fondo tarda en ejecutarse 30 µs, cuenta con 3 interrupciones cuyos máximos tiempos de ejecución son de A = 150 µs, B = 320 µs, y C = 120 µs, y sólo se deshabilitan las interrupciones al principio de la rutina de fondo antes de entrar en el bucle principal. Si el tiempo empleado en salvar los registros en pila para atender a una interrupción es de 16 µs, calcule el Tiempo de respuesta en el peor de los casos para una cuarta interrupción D que se escribiera y fuera menos prioritaria que las anteriores. Se asume que la máxima frecuencia de petición a interrupción es de 100Hz para cualquiera de ellas.**

30 + 150 + 320 + 120 +16 = 636 µs

Las instrucciones que afectan a la hora de retardar la atención a interrupción son las que puedan estar siendo procesadas cuando se produzca la IRQ. Si las otras interrupciones son más prioritarias no serán desalojadas por esta.

2.2.3.2 Interrupciones no Enmascarables

Como se ha comentado, en la mayoría de los procesadores, se puede habilitar y deshabilitar ciertas interrupciones, denominadas enmascarables. Esto permite controlar el flujo de eventos hacia el sistema. Muchos sistemas tienen además las llamadas *Interrupciones No*

[3] En este contexto entenderemos por *rutina de fondo* a cualquier rutina no vinculada directamente con una interrupción, que realiza un procesamiento de baja prioridad o de poca importancia relativa, y que es susceptible de ser interrumpida sin problemas. Las rutinas de fondo no tendrán por tanto restricciones de respuesta en tiempo real, y sus funciones habituales son las de procesamiento puro de información, sondeo de dispositivos, o simplemente la espera de interrupciones.

Enmascarables (NMI), que no se pueden deshabilitar por software. Estas pueden ser usadas bajo cualquier circunstancia, como por ejemplo para contabilizar correctamente el tiempo. Tampoco se deben poder deshabilitar las interrupciones vinculadas a mecanismos de seguridad críticos para el sistema, como la notificación de avería del reloj, tensión de alimentación baja, petición de *RESET*, *COP*, *Watch Dog*,… Por lo general las NMI no son tan frecuentemente llamadas como las interrupciones enmascarables, pero es muy importante contar con ellas para poder contemplar situaciones excepcionales de importancia.

2.2.3.3 Pulso de Reloj

El pulso de Reloj es el latido del sistema. Es producido habitualmente por un circuito resonador basado en un cristal de cuarzo. Mediante éste los sistemas sencillos mantienen el tiempo relativo y absoluto del sistema. Los sistemas más complejos pueden contar con una circuitería adicional dedicada a almacenar un reloj de tiempo real, que cuente con su propio cristal de cuarzo. El pulso de reloj permite implementar temporizadores, muy útiles en los STR para generar interrupciones a intervalos programables. Para poder desarrollar un SOTR se debe emplear al menos una interrupción de reloj.

2.2.4 Niveles de memoria

La memoria es uno de los recursos más característicos de cualquier sistema informático. Una parte importante de los SS.OO. se encarga de su correcta administración para satisfacer las demandas de las tareas. Por razones prácticas, los sistemas informáticos modernos utilizan un esquema de memoria dividido en varios niveles con el objetivo de poder disponer de mucha memoria para programas y datos, manteniendo un tiempo medio de acceso pequeño.

La **Figura 2-3** muestra los distintos niveles de memoria que suelen utilizar los ordenadores de propósito general para poder procesar grandes cantidades de información lo más rápidamente posible.

El objetivo final es que los datos se encuentren en los registros de la CPU en el momento en que se requiera su procesamiento. Para ello se considera el tiempo de acceso, que es el requerido en traer un dato a la CPU desde un almacenamiento externo.

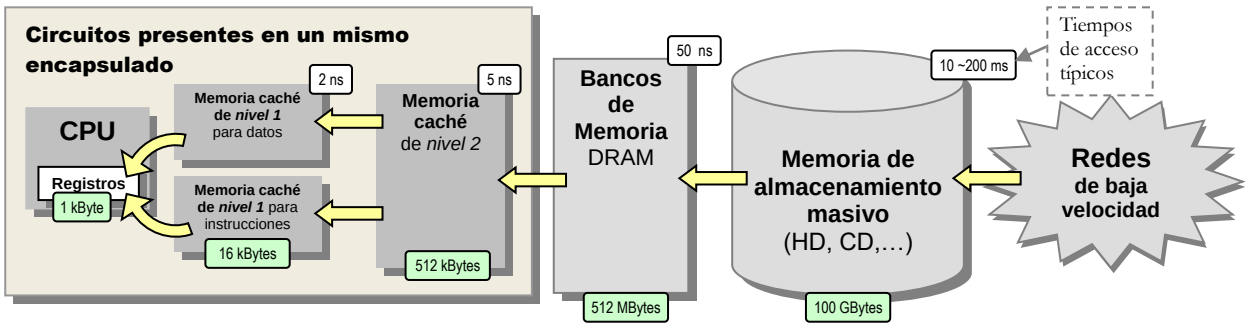


Figura 2-4. Niveles de memoria de una computadora de propósito general

Cada vez que la CPU necesita acceder a una posición, se consulta en la caché de nivel 1 para intentar usarla desde ahí. En caso de no encontrarla, se consulta al resto de niveles de memoria hasta encontrarla. Si no se encontrara en la memoria RAM, un mecanismo software la buscaría en el disco duro o en la red, comportándose por tanto como niveles de memoria adicionales. Una vez encontrado, se propaga a los niveles de memoria inferiores quedándose copiado para posteriores accesos. Este mecanismo aumenta la probabilidad de encontrar la información que es accedida con más frecuencia en memorias más rápidas.

Para acelerar la velocidad media de ejecución manteniendo unos costes competitivos, se han realizado multitud de estudios sobre la forma que tienen los sistemas informáticos de acceder a la memoria. Los resultados han desvelado algunas conclusiones interesantes, que se podrían resumir en dos:

- Los accesos a memoria tienen mucha *coherencia espacial*.- La probabilidad de acceder a datos que estén en posiciones próximas es muy alta. Encontramos un ejemplo de esto en los programas en ejecución, que leen secuencias de instrucciones consecutivas según las van procesando. Otro caso habitual se da al recorrer secuencialmente un array para realizar un procesamiento con sus elementos. Ambas actividades se realizan con accesos de carácter secuencial y hasta cierto punto previsible, en los que el espacio entre datos es muy pequeño.
- Los accesos a memoria tienen mucha *coherencia temporal*.- La probabilidad de acceder a posiciones que hayan sido utilizadas recientemente es muy alta. Esto se da por ejemplo en el acceso a variables que acumulan resultados generados en puntos próximos de un mismo programa. Otro ejemplo lo encontramos en la ejecución de bucles, o al llamar frecuentemente a las mismas subrutinas. En ambos casos se vuelven a procesar instrucciones y datos que se habían utilizado en un pasado reciente.

Las memorias caché vinculadas al procesador, también llamadas caché hardware de la CPU, se emplean para almacenar información que se supone va a ser utilizada repetidas veces en un futuro cercano. Para ello se benefician del hecho de que estadísticamente exista una gran coherencia temporal tanto del código como de los datos sobre los que se opera.

Las caché tan solo almacenan copias de zonas de memoria correspondientes a las memorias de nivel inmediatamente superior.

Si ha pasado mucho tiempo desde la última vez que se accedió a un dato presente en caché, será desechado y reemplazado por otra dato al que se estime se va a acceder con más probabilidad. Si se hubiera modificado el dato que se va desechar, entonces se actualizará la memoria de nivel inmediatamente superior con su contenido.

Las memorias caché están fabricadas con celdas de almacenamiento estático, por lo que sus tiempos de acceso son del orden de nanosegundos. Su funcionamiento es bastante transparente al programador. En los micros más recientes se han incorporado algunas instrucciones para poder controlarlas desde el modo usuario, lo que permite aumentar notablemente la velocidad media de ejecución de las aplicaciones que las utilicen.

La memoria RAM tiene gran capacidad, pero como contrapartida está fabricada con celdas de almacenamiento dinámico que son más lentas que las SRAM. El tiempo de acceso de las DRAM es del orden de decenas de nanosegundos.

Los sistemas de almacenamiento masivo como el disco duro o el CD-ROM, presentan unos tiempos de acceso del orden de decenas de milisegundo, muy por encima del que presentan las memorias de semiconductores. Esto es debido fundamentalmente a que cuentan con piezas mecánicas para realizar la E/S.

Debido al diseño electrónico, la granularidad en el acceso a memoria es más fina (selectiva) cuanto más cercano se esté de la CPU. En la lectura de información desde la RAM[4], se recuperan bloques mayores que los que solicita la CPU en principio, por ejemplo en la caché de nivel 1 la mínima cantidad de memoria direccionable suele ser del orden de 8 bytes mientras

[4] Como ya sabemos, RAM significa *Memoria de Acceso Aleatorio*. Téngase en cuenta que incluso la *Memoria de Sólo Lectura* o ROM construida con semiconductores, también suelen ser de acceso aleatorio (no es así en las ROM serie). La memoria que almacenan las cachés hardware no es directamente direccionable, por lo que aunque se utilicen celdas estáticas de almacenamiento de bits similares a las de las SRAM, no se trata realmente de memora RAM.

que en un disco duro suele ser de 4096 bytes. Esto ocurre de manera aún más acusada en el acceso a los datos de los sistemas de almacenamiento masivo, que son gestionados como bloques indivisibles de varios cientos o miles de bytes. Debido a la coherencia espacial, no es un problema demasiado grande, ya que todos esos datos que se recuperan de más, a menudo terminan siendo utilizados posteriormente eliminando la necesidad de realizar nuevas peticiones.

Se suele considerar que la memoria de los registros de la CPU es de nivel 0, y la memoria RAM es de nivel 3. Como nivel adicional, se podría mencionar la información almacenada por otros sistemas informáticos, que se encuentra accesible mediante interconexión por redes.

Apréciese que cada nivel de memoria se comporta como una caché para los niveles superiores.

Las flechas de la figura representan el flujo de información cuyo destino último es la CPU en donde podrá ser procesada. Estas parten desde los dispositivos más lentos, y van hasta los registros del procesador. Los buses son más rápidos cuanto más cercanos estén de la CPU, a la par que son menos estándares.

Cada vez que se requiere un dato que no está presente en los registros de la CPU, el sistema comienza un ciclo de búsqueda en cascada interrogando a cada uno de los niveles de memoria hasta dar con él. Cada consulta infructuosa implica la acumulación de una penalización temporal en la búsqueda del dato.

A continuación se muestra un programa que se beneficiaría enormemente del uso de las memorias de nivel 1 y 2:

```
#include <memory.h>

//------------------------ EstadisticaDeNotas -----------------------------
// Calcula la función de distribución de las calificaciones de los alumnos.
// Entradas:
//    notas - Array con todas las notas que se van a considerar.
//    numNotas - Número de alumnos.
//    fdd - dirección de un array de 11 elementos (de 0 a 10) que será
//          rellenado con la función de distribución.
// Comentario: Por claridad no se incorpora código de detección de errores.
void EstadisticaDeNotas(const int notas[], int numNotas, int fdd[])
{
    const int *itNotas = notas; // Iterador de notas

    memset(fdd, 0, 11 * sizeof(fdd[0])); // Limpia el array de la FDD

    while(numNotas--)
    {
        int nota = *itNotas++;

        if(nota >= 0 && nota <= 10)
            fdd[nota]++;
    }
}

int main(int argc, char* argv[])
{
    static int notas[500];
    int fdd[11];

    // Se rellena el array con las notas de 500 alumnos
    // . . .

    EstadisticaDeNotas(notas, 500, fdd);

    // Se hace uso de la información obtenida
    // . . .

    return 0;
}
```

Programa 2-1. Uso automático de las memorias caché.

La parte más interesante está en el acceso a memoria que se realiza desde el bucle `while` que se incluye en la función `EstadisticaDeNotas()`: El acceso al array `notas[]` tiene gran coherencia espacial, mientras que tanto el código máquina del bucle como los datos que forman el array de la función de distribución `fdd[]`, tienen además coherencia temporal.

En la **Figura 2-5** se muestra una representación gráfica del acceso a memoria que realiza el programa.

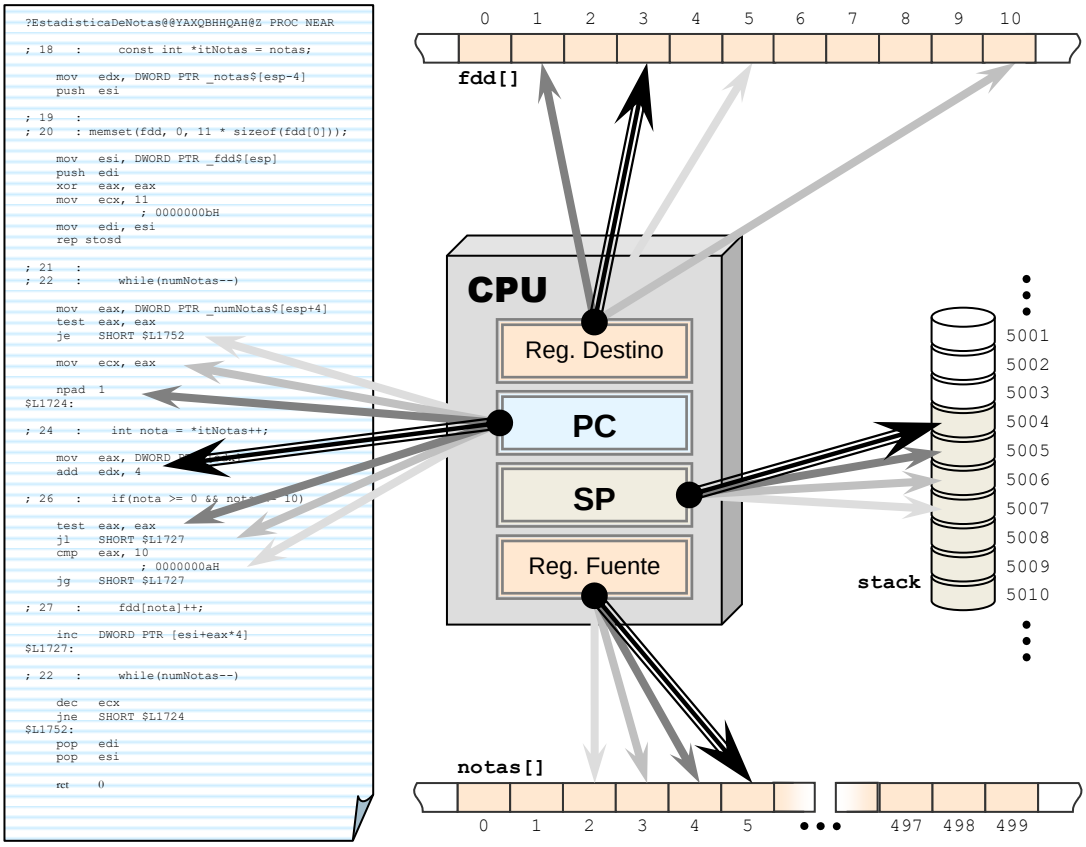


Figura 2-5. CPU accediendo a diferentes partes de la memoria

El código máquina que es recorrido mediante el registro contador de programa (PC), se almacena en una parte de la caché de nivel 1 destinada a almacenar exclusivamente instrucciones. El resto de información se almacena en la otra parte de la caché de nivel 1 destinada exclusivamente a datos.

Ejercicios

4. ¿Pueden existir tareas que utilicen el mismo código máquina?

Sí. Pero no deben compartir datos sin una sincronización o acuerdo entre ellas.

5. Justifique por qué las memorias caché aceleran las operaciones con la pila.

El acceso a memoria que se realiza con la pila tiene mucha:

- **Coherencia temporal**.- Debido a que las tareas utilizan intensivamente la pila, la probabilidad de consultar o actualizar posiciones recientemente accedidas es muy alta.
- **Coherencia espacial**.- La propia estructura de pila (LIFO), y la forma en que es utilizada implica que se accedan a datos próximos a los más recientemente accedidos.

6. Cite factores que remarquen el contraste que impone la proximidad de los sistemas de almacenamiento a los registros de la CPU.

Según se esté mas cerca de la CPU, los dispositivos de almacenamiento son:

- Más caros, atendiendo al precio por bit.
- Más pequeños, tanto en tamaño físico como en capacidad de almacenamiento.
- Más rápidos y sofisticados.
- Más fiables.
- Más selectivos con la información que manejan.
- Menos normalizados, al igual que los buses que se emplean para comunicarlas entre sí.

7. Analice el siguiente programa y conteste justificadamente a las siguientes cuestiones ¿Están optimizando las cachés el acceso que se hace desde el bucle `while` al array `acumula[]`? En un programa cualquiera, ¿dónde es más fácil encontrar coherencia espacial: en código o en datos?

```
#include <stdlib.h>

int main(int argc, char* argv[])
{
    int num = 100;
    int acumula[1000];

    memset(acumula, 0, sizeof(acumula)); // Limpia el contenido del array

    while(num--)
    {
        int pos;

        pos = rand() % 1000; // Calcula un número aleatorio de 0 a 999

        acumula[pos]++;
    }

    return 0;
}
```

Al acceder a posiciones aleatorias del array, las cachés no se pueden beneficiar significativamente de ningún tipo de coherencia.

La ejecución de código conlleva implícitamente coherencia espacial, ya que entre salto y salto, las instrucciones se encuentran consecutivas en memoria, y a menudo muchos bucles internos están formados por apenas unas pocas instrucciones consecutivas. Como se demuestra en este ejemplo, esto no es necesariamente cierto en el acceso a los datos.

8. En la Figura 2-4 se muestra una memoria de nivel 1 dividida en dos: una para instrucciones y otra para datos. ¿A qué cree que puede ser debido?

La forma de acceder tanto al código de los programas como a los datos que manejan tiene mucha coherencia espacial y temporal. No obstante ambos son coherentes únicamente con información de su mismo tipo (instrucciones con instrucciones y datos con datos), almacenándose incluso en distintas zonas de la RAM. El uso de cachés específicas mejora el aprovechamiento por coherencia espacial de las dos. Además este tipo de memoria basada en arquitectura Hardvard aumenta el rendimiento total al comunicarse con el procesador mediante dos buses independientes de instrucciones y datos.

9. ¿Qué repercusiones tiene el uso de diferentes niveles de memoria para la realización de un sistema de tiempo real?

Pese a que la división de la memoria en niveles permite disponer de grandes cantidades de memoria económica a la que se accede con unos tiempos medios aceptables, los tiempos máximos pueden llegar a ser prohibitivamente altos para el desarrollo de STR.

Por otra parte el uso de piezas móviles en los sistemas de almacenamiento masivo disminuye notablemente su fiabilidad, y aumentan su consumo.

No obstante, si sólo se empleara la memoria que está más allá de la RAM en tareas no críticas, se podría admitir su uso, de lo que se beneficiarían las tareas interactivas que requieran grandes cantidades de datos.

10. Escriba un programa en el que el acceso a datos se realice con mucha coherencia temporal, pero sin coherencia espacial.

```
int main(int argc, char* argv[])
{
    int num = 100;
    static int acumula[1000];

    while(num--)
    {
        // Acceso frecuente a los mismos datos distantes entre sí
        acumula[0]++;
        acumula[100]++;
        acumula[30]++;
        acumula[550]++;
        acumula[900]++;
        acumula[770]++;
    }

    return 0;
}
```

11. ¿Qué es el tiempo de acceso[5] total de los registros de la CPU?

El tiempo de acceso que se considera en un sistema informático es el empleado en traer algún dato a los registros de la CPU. Si ese dato ya está allí, su tiempo de acceso es nulo.

[5] Como factor tecnológico, el tiempo de acceso a una memoria se considera al que transcurre desde que se le solicita un dato hasta que es colocado en el bus. Desde un punto de vista más informático, se le considera como el tiempo necesario desde que la CPU hace la solicitud hasta que llega a sus registros internos.

2.3 Sistemas Operativos para aplicaciones industriales

Los SS.OO. de propósito general son los que usan los computadores de sobremesa y los portátiles. Están diseñados para proporcionar una amplia gama de servicios a las aplicaciones y por consiguiente a los usuarios. Intentan minimizar el tiempo medio de ejecución de los programas, y muy a menudo el tiempo de [primera] respuesta para ofrecer resultados interactivos. No están diseñados para ejecutar un tipo de programa en particular, aunque estos suelen estar orientados hacia la administración y desarrollo de documentos de usuario. La información procesada es de diversa naturaleza, y por lo general los programas que ejecutan son heterogéneos y complejos. Por ejemplo, un simple procesador de textos como **MS Word**, es desde el punto de vista algorítmico más complejo que el programa que controle los mecanismos de la más potente central nuclear, aunque por otra parte no es tan fiable.

Estos SS.OO. pueden llegar a ser muy grandes y complejos, lo que los convierte en impredecibles y poco fiables, siendo este un gran inconveniente para emplearlos en aplicaciones de Tiempo Real. Así mismo el desarrollador no puede tener una visión clara de todos y cada uno de los aspectos que pudieran influir en el comportamiento final de su aplicación, lo que añade aún más incertidumbre a los tiempos de respuesta de las tareas. Esto presenta una problemática a resolver nada trivial, si lo que pretendemos es que el S.O. sea capaz de responder adecuadamente a las demandas de fiabilidad y determinismo que requieren las tareas de TR. La programación de TR podría ser entendida como un conjunto de ideas, conceptos y técnicas que nos permiten dividir problemas reales en unidades de código que deben ser procesadas respetando y teniendo en cuenta el paso del tiempo en el entorno real externo al sistema, o bien como eventos que llevan a una tarea de un estado a otro, produciendo las salidas correspondientes.

La respuesta a todo esto se basa en hacer un S.O. específico que de solución a las necesidades concretas de las aplicaciones de automatización y control. Con este propósito surgen los llamados Sistemas Operativos de Tiempo Real (S.O.T.R.), concebidos para ejecutar tareas con restricciones de tiempo real.

El S.O.T.R. se diseña para alcanzar una elevada integración con la máquina sobre la que va a correr, incorpora sólo los componentes esenciales para su fin, evitando incluir código que no vaya a ser útil, por lo que suele ser comparativamente más pequeño y compacto que un S.O. para ordenadores de uso genérico. Está destinado principalmente a dar soporte para la ejecución de aplicaciones de control simples y monótonas, que durante la vida útil del sistema son actualizadas en escasas ocasiones o nunca. Por lo general no da soporte al desarrollo de programas sino únicamente a su ejecución, necesitándose una computadora adicional que cuente con las herramientas de desarrollo, y depuración necesarias. En el caso de desarrollar para microcontroladores y DSP, es necesario el uso de una computadora de propósito general en la que se puedan generar los programas, simularlos, y depurarlos. La depuración se realiza bien con ayuda del simulador, o mediante depuración remota, en caso de usar un depurador residente en el controlador.

Como ya sabemos, una de las principales características de un S.O.T.R. es su determinismo temporal, lo que se traduce en que es posible conocer el tiempo máximo de ejecución de las tareas a las que da soporte.

En la **Figura 2-6** se muestra estadísticamente la diferencia entre los tiempos de ejecución de una misma tarea que usa intensivamente la CPU, ejecutándose sobre un S.O. de propósito

general y sobre un S.O.T.R.. En ambos casos se ha realizado la ejecución del programa numerosas veces y en diferentes circunstancias de carga del sistema, evaluado el tiempo que transcurre desde que se da la orden de ejecución, hasta que se devuelven los resultados y finaliza, a lo que también se llama *tiempo de respuesta*.

En aplicaciones de respuesta interactiva existe el concepto de *tiempo de primera respuesta*. Es el que transcurre desde que empezó a ejecutarse la tarea hasta que comienza a generar los resultados. En tal caso se supone que va a estar un cierto periodo de tiempo ofreciendo más datos.

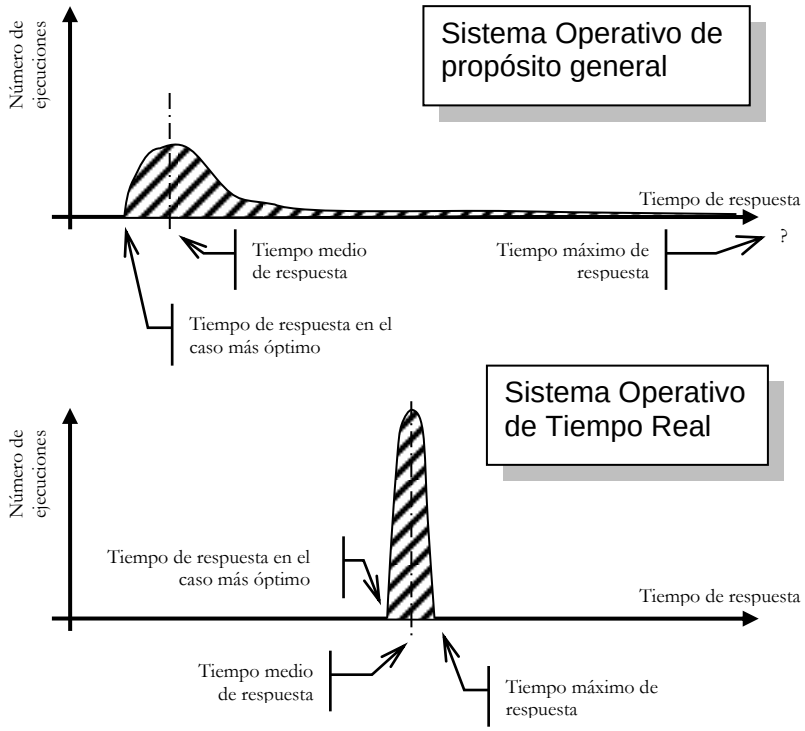


Figura 2-6 Estadística de tiempos de ejecución de una misma rutina

El tiempo medio ha sido menor en el primer caso ya que la arquitectura está optimizada para ello: el hardware cuenta con memorias caché, controladores de DMA[6] (*Direct Memory Access*), CPU con ejecución de instrucciones fuera de orden, buses de alta cadencia,... Además estos sistemas gestionan una gran cantidad de periféricos que a menudo incorporan sus propias memorias y procesadores que, pese a descargar a la CPU de tareas de control específico, pueden requerir su atención en cualquier momento. Por otra parte el gran número de programas ejecutándose, aunque se trate tan solo de los correspondientes al S.O. y a la interfaz de usuario, la administración de grandes cantidades de memoria dividida en niveles, y otros factores convierten al sistema en complejo e impredecible. Los tiempos de ejecución sólo son calculables en la práctica desde un punto de vista estadístico. Como resulta por la gráfica, el tiempo máximo que se obtendría en la peor circunstancia posible, puede ser grandísimo, ya que estos sistemas son capaces de posponer la ejecución de un programa atendiendo a políticas de planificación internas. Lamentablemente este tipo de sistemas tampoco están exentos de "cuelgues" esporádicos debidos a errores de *software* o al uso de un *hardware* de baja calidad.

En el dibujo que aparece en la parte inferior de la **Figura 2-6**, el tiempo medio de respuesta es muy superior, ya que por lo general no se hace uso de hardware avanzado y se

[6] El DMA es un elemento muy común en los sistemas informáticos avanzados. Es un circuito que se utiliza para copiar bloques de memoria de un sitio a otro funcionando en paralelo con la CPU.

prescinde en la medida de lo posible de memorias dispuestas en varios niveles. El reloj que marca la frecuencia de funcionamiento no suele estar tan forzado por razones de fiabilidad y consumo. Sin embargo ya que el sistema en conjunto es sencillo, el tiempo máximo de respuesta es mucho más fácilmente calculable.

En ambos casos existirá un tiempo mínimo de respuesta, que se dará cuando todas las circunstancias hayan sido propicias, por lo que puede ser estimado con relativa facilidad.

Cabe destacar además que uno de los posibles usos de un S.O.T.R. es el de ofrecer una base sobre la que se puedan ejecutar programas realizados usando los llamados lenguajes de tiempo real como **Ada**, **Modula 2/3**, etc. ya que si fueran ejecutados sobre otro tipo de S.O. no podrían desplegar sus avanzadas características.

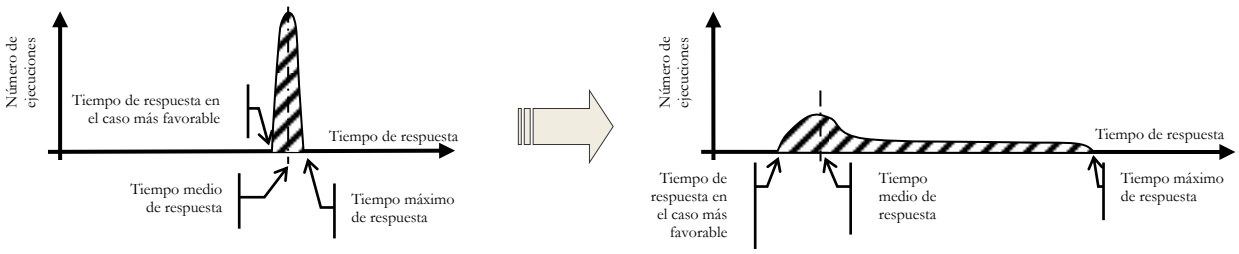
Ejercicios

12. ¿Por qué una tarea de tiempo real no haría uso directo de un disco duro o de una Internet? Si necesitara datos de alguno de ellos, ¿Cómo los obtendría entonces?

Porque los tiempos de acceso son indeterminados siendo los máximos muy elevados. Además el hecho de usar piezas mecánicas sujetas al desgaste y roturas -en el caso del disco duro-, y el indeterminismo de las redes de Internet habituales, los convierten en sistemas muy poco fiables en comparación con las memorias de semiconductores.

Para poder utilizarlos se podría hacer uso de otra tarea no crítica, comunicándolas mediante el uso de memoria compartida como una FIFO software. En el caso de que la utilidad de la memoria de almacenamiento masivo sea guardar datos de sólo lectura, el ingeniero debería plantearse la posibilidad de usar memorias no volátiles de gran capacidad como las Flash, mucho más deterministas y fiables que los discos.

13. A continuación se muestran las funciones de distribución de los tiempos de ejecución de una tarea antes y después de habilitar la caché hardware de un sistema. Indicar en qué aspectos ha influido su uso para provocar este cambio.



Las caché están diseñadas para disminuir el tiempo medio de acceso a la información que almacena la RAM. La probabilidad de encontrar en ellas la información que se va a necesitar en un futuro próximo es muy alta, por lo que debido a que son mucho más rápidas que la RAM, hacen que aumente la velocidad de ejecución. Sin embargo si no se encuentra el dato buscado, se transmite la petición al siguiente nivel de memoria, proceso que en el peor de los casos ralentiza la operación mucho más que si no se hubiera empleado esa caché.

El número de ejecuciones no decrece asintóticamente por el simple hecho de usar cachés como ocurre en la **Figura 2-6**, sino que el tiempo máximo de acceso es conocido. El decrecimiento asintótico se produce por el uso de sistemas de almacenamiento que

incorporan piezas móviles que les hacen poco fiables, y también por el uso de SS.OO. complejos con demasiados puntos oscuros poco depurados.

No es una decisión descabellada deshabilitar las cachés en los STR para poder conocer los máximos tiempos de respuesta de forma más determinista, con la consiguiente caída generalizada de rendimiento.

2.3.1 Clasificación de SS.OO.

El S.O.T.R. pretende ser el soporte a un tipo de programación en el que se tiene varias tareas que realizar, unas periódicas y otras no, y que deben responder a eventos externos en franjas de tiempo bien definidas.

Hay varias formas de llevar a cabo esta función, dependiendo de las necesidades de la aplicación:

2.3.1.1 Máquina desnuda

Aquí se engloban todos los diseños carentes de Sistema Operativo, como son la inmensa mayoría de los que están basados en dispositivos programables pequeños. Son útiles cuando no existen complicaciones *software*, y las tareas a realizar son muy sencillas, o bien cuando están especializados en un tipo de procesamiento muy específico.

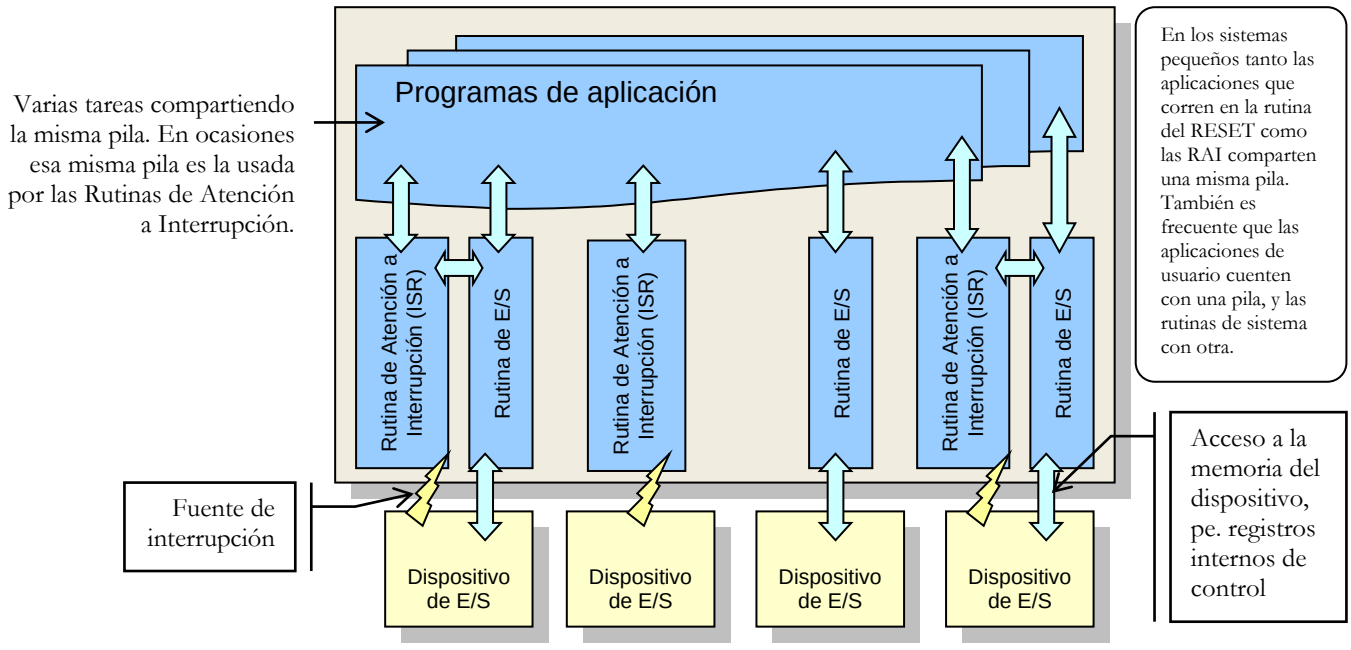


Figura 2-7. Sistema computador basado en una máquina desnuda

El programador a menudo pone el código de respuesta a eventos en las propias rutinas de tratamiento a interrupción, aumentando consiguientemente el tiempo en que la máquina permanece ininterrumpible. También se suele crear una rutina con un bucle infinito para ejecución de fondo con los quehaceres de respuesta interactiva, es decir aquellos que necesitan una ejecución media lo más rápida posible. Se suelen programar en C o directamente en Ensamblador.

Por lo general, estos sistemas no emplean ningún tipo de protección de memoria ni recursos a excepción de la que presenta la propia ROM, por tanto las tareas son libres de

utilizarlos desde cualquier parte del código. Cada subprograma tiene el poder de la máquina durante el tiempo en el que está ejecutando su código.

En la **Figura 2-7** se muestra esquemáticamente un sistema computador basado en máquina desnuda. El programador de la aplicación es el encargado de escribir todas las rutinas, ya sean ISRs, subprogramas de Entrada/Salida, rutinas de cómputo, etc. Existe comunicación directa entre todas las partes del sistema, lo que permite que las subrutinas sean llamadas indistintamente desde cualquier parte. Es habitual que todas las tareas compartan la misma pila.

En la figura aparecen varios dispositivos de E/S, dos de los cuales generan interrupciones, y pueden ser accedidos como posiciones de memoria (temporizadores, memorias externas: de puerto dual, *fifos*,…). También figura un dispositivo que sólo genera interrupciones (pe. pulsadores, interruptores, finales de carrera, *bumpers*, bigotes de gato,…) y otro que sólo es accedido mediante lectura / escritura directa en sus registros (pe. dispositivos accesibles sólo por sondeo: conversores AD/DA sin buffer,…).

2.3.1.2 Sistema Operativo monolítico

Además de las rutinas que se ejecutan en respuesta a las interrupciones, a menudo los sistemas cuentan con un conjunto de funciones con interfaces bien definidas no necesariamente estructuradas que pueden ser llamadas libremente las unas a las otras, y están encargadas de realizar funciones de bajo nivel, relacionadas con la gestión de recursos, y la interfaz con el hardware. Estas rutinas se ejecutan con el nivel de privilegios hardware más elevado que se cuente, y forman el llamado *núcleo* del Sistema Operativo.

La forma que tienen las aplicaciones de solicitar acceso al hardware es mediante las llamadas *llamadas al sistema*. Estas se realizan mediante una interrupción software, que al cambiar automáticamente a modo núcleo, permite ejecutar las rutinas de bajo nivel del S.O. El programador en la práctica percibe las llamadas al sistema como una biblioteca de funciones de bajo nivel, que a menudo está oculta tras otra biblioteca de alto nivel. En el caso de que no se pueda interrumpir al núcleo, las llamadas se ejecutarían de manera indivisible o atómica, llamándose entonces núcleo de tipo *monitor monolítico*. En los sistemas más primitivos la mayor parte del código del núcleo se ejecuta exclusivamente en respuesta a las interrupciones y a las llamadas al sistema. En los más evolucionados, el núcleo cuenta con sus propias tareas que se ejecutan con mayor prioridad que las del usuario (*kernel tasks*).

Puede que sea necesaria la existencia de protección entre recursos. En tal caso el núcleo los asigna dinámica o estáticamente a cada una de las tareas, y se encarga de que las demás no interfieran durante el tiempo en que sean utilizados. Como ya se comentó la mejor forma de conseguir esto es, mediante el uso de una CPU que permita conmutar entre un funcionamiento en *modo usuario*, y otro *modo núcleo*. En modo usuario, los programas no pueden acceder a todos los recursos de la máquina, sobre todo a los más críticos; no pueden por ejemplo deshabilitar las interrupciones de forma global. El modo núcleo por el contrario tiene acceso a todos los recursos e instrucciones del procesador, siendo el idóneo para ejecutar las rutinas del propio núcleo.

El hecho de que un sistema sea *monolítico* no implica que no tenga un diseño interno modular y bien establecido, ni que no se puedan incorporar o retirar controladores de dispositivo en tiempo de ejecución. Tan solo significa que las tareas del núcleo no tienen protección de acceso entre sí.

Recordemos que si un procesador no incorpora características de protección, es equivalente a que siempre ejecutara el código en modo núcleo, en tal caso, hasta la tarea más insignificante podría hacerse con el control total del sistema. Es posible no obstante utilizar

hardware adicional de protección, basado normalmente en restricciones de acceso a zonas de memoria críticas.

Para aligerar las rutinas de tratamiento de interrupción, se traslada la mayor parte del código de las ISR a las rutinas del sistema.

En este tipo de sistemas cada tarea utiliza su propia pila. El núcleo lleva un control de tanto de tareas como de recursos.

La **Figura 2-8** muestra el esquema de un sistema de un núcleo monolítico simple.

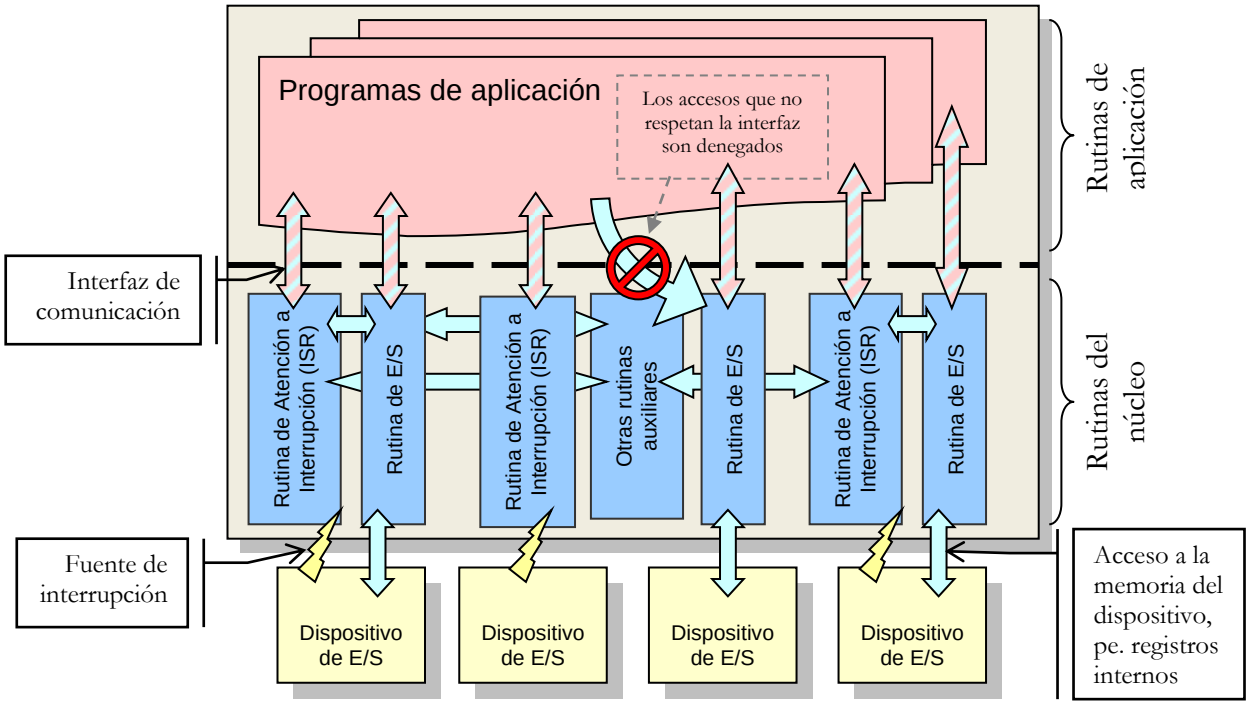


Figura 2-8. Sistema Operativo monolítico simple

En la figura aparecen dos zonas claramente delimitadas: la superior con los programas de aplicación, y la inferior con las rutinas del monitor. Las aplicaciones tan sólo son capaces de comunicarse con el núcleo y en última instancia con el *hardware* a través de una interfaz de comunicación bien definida. Cualquier intento de acceso desde la capa de aplicación a las rutinas del núcleo que no respete la interfaz, será rechazado si la máquina implementa protección entre áreas de memoria. Dentro del núcleo las rutinas se pueden llamar unas a otras libremente y utilizar estructuras de datos compartidas, sin la necesidad de jerarquías o interfaces internas normalizadas. Es un núcleo con una concepción no modular.

Se da soporte a la multitarea: el codificador crea diferentes programas, cada uno para una necesidad diferente, y el núcleo se encarga de ejecutarlos de acuerdo a su prioridad, y a la política de ejecución elegida.

Además estos sistemas a menudo dan soporte a la multiprogramación, que consiste en la conmutación forzosa dirigida por el núcleo del uso de la CPU entre las tareas.

2.3.1.3 El desarrollo en capas del S.O.

Debido a la creciente demanda de complejidad en los SS.OO., pronto se hizo necesario el uso de técnicas avanzadas de diseño que permitieran desarrollar núcleos más grandes y seguros por los equipos de desarrolladores. Una de las ideas más básicas es desarrollar software modular en base a interfaces normalizados. Esto permite que cada grupo de desarrollo se

concentre sólo en la parte que le atañe, sin tener que preocuparse excesivamente por las implicaciones que tendrían los mecanismos que se implementaran en otros módulos.

En esta línea surge el *desarrollo en capas* que consiste en la creación de módulos de software independientes interconectados entre sí. Cada módulo incorpora los programas necesarios para dar respuesta a distintos requerimientos como son la gestión de: tareas primitivas, tareas clientes, memoria, archivos y dispositivos, entre otros. Se trata de que cada uno de ellos conforme una etapa del núcleo, estando el hardware en la parte más inferior, y las aplicaciones de usuario en la más superior, tal como muestra la **Figura 2-9**. En este caso, la única forma de acceder a las capas más internas es a través de las capas intermedias:

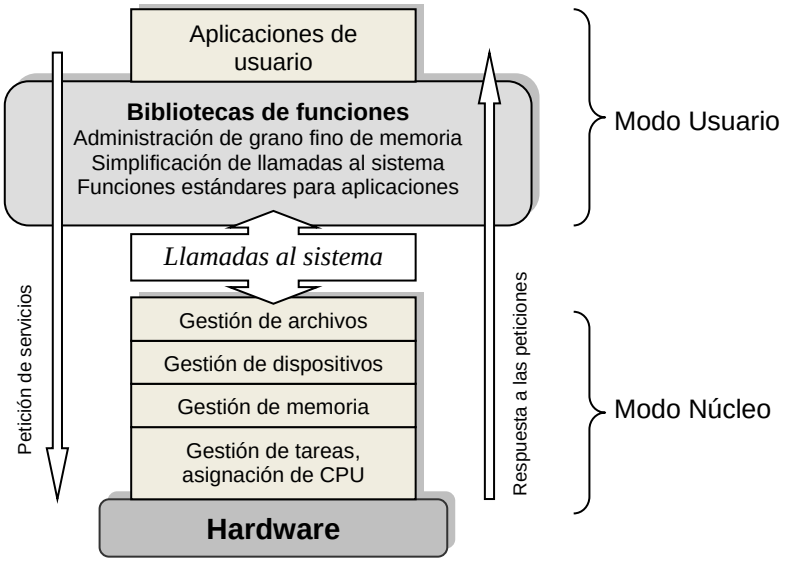


Figura 2-9. Sistema Operativo organizado por capas

Según la **Figura 2-9** cualquier petición que quisiera hacer una aplicación de usuario debería recorrer todos los niveles necesarios hasta llegar al módulo que la pueda atender. Si fuera necesaria, la respuesta debería volver desde allí de nuevo hasta la aplicación. Las llamadas al sistema son la interfaz que tienen las aplicaciones para comunicarse con el núcleo. El conjunto de todas las llamadas al sistema se percibe en la práctica como una biblioteca de funciones más. No todos los accesos al hardware han de realizarse con la intervención expresa del núcleo, sino que en ocasiones a las aplicaciones de usuario se les permite acceder directamente a los dispositivos, una vez el S.O. lo consienta.

La interfaz del núcleo suele estar en muchas ocasiones oculta a las aplicaciones de usuario por bibliotecas de funciones que se encuentran igualmente a nivel de usuario, tal como muestra la figura. Se suele decir entonces que tenemos una biblioteca envoltorio (*wrapper*) de otra que es habitualmente de más bajo nivel. Por ejemplo la función `printf()` de la biblioteca estándar de C realiza internamente una llamada al sistema `write()` o similar, que es la encargada real de acceder en última instancia al hardware de vídeo.

2.3.1.4 Enfoque Micronúcleo *(µ-kernel)*

El desarrollo en capas conlleva una serie de inconvenientes: debido a que dependen unas capas de las otras, se impide que se puedan hacer modificaciones de importancia sin que se afecte de alguna forma a las demás. El elevado número de interacciones existentes entre capas

contiguas hace que cuenten con una interfaz excesivamente férrea, difícil de mantener y actualizar. Además la abundancia de estratos puede ralentizar la velocidad de respuesta.

Este panorama sugiere la necesidad de crear un modelo de núcleo distinto, en el que la cohesión entre sus distintas partes sea la menor posible, para evitar problemas de interdependencias. Se libera del núcleo a las rutinas menos esenciales, haciéndolas ejecutarse bien en tareas del sistema a nivel de núcleo o bien como tareas a nivel de usuario. Esto proporciona más seguridad al sistema, ya que un error en cualquiera de estas tareas permanecerá completamente estanco, y no tendrá comportamientos colaterales de gravedad. Tan sólo la parte más esencial del núcleo cuenta con privilegios totales de acceso. Es la encargada de asignar tiempo de CPU a las distintas tareas, ocupándose de todos los elementos necesarios para conmutar entre ellas. Esta parte se llama *micronúcleo*, ya que su tamaño es mucho menor que el de los núcleos tradicionales.

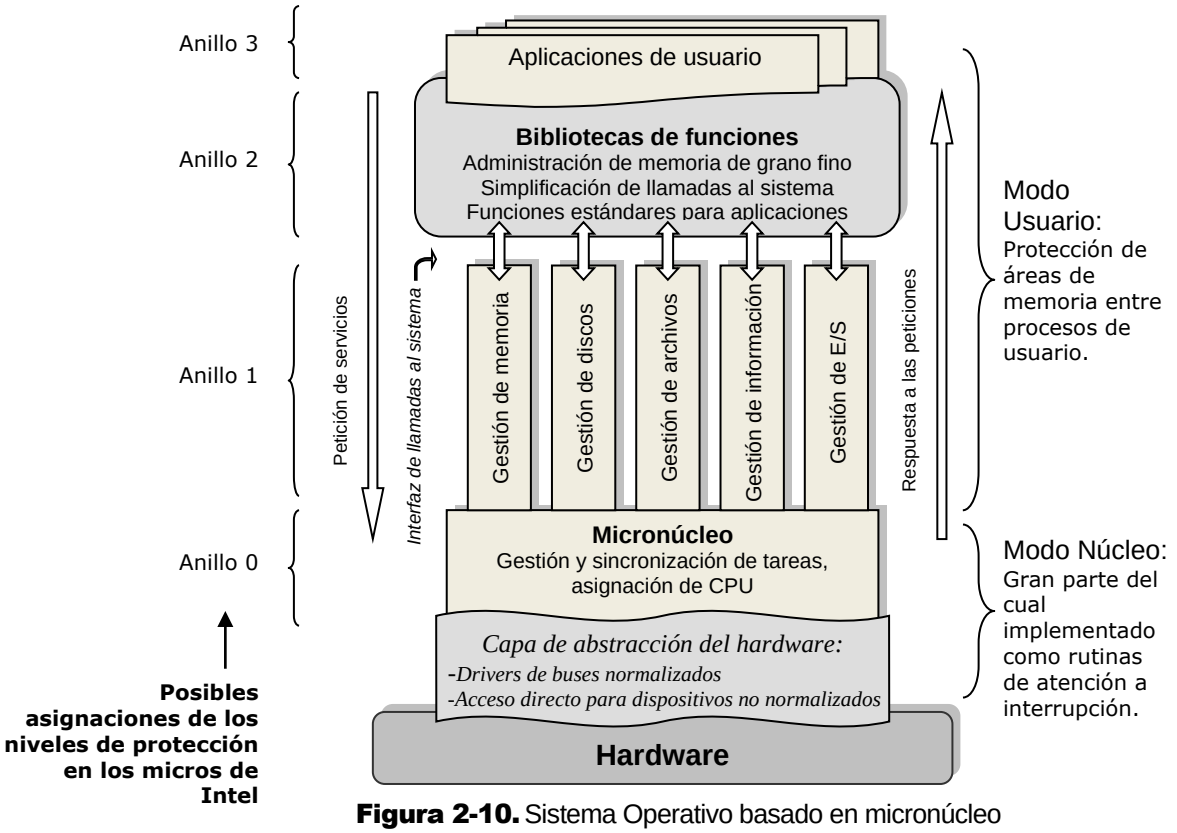


Figura 2-10. Sistema Operativo basado en micronúcleo

En la Figura 2-10 aparece el esquema de un sistema micronúcleo. Como se puede apreciar los módulos que aparecían de forma estratificada en el modelo de núcleo por capas, han adoptado una estructuración vertical. Los diferentes módulos aparecen aislados del hardware al que no pueden acceder directamente como antes, ya que entre otras cosas no tienen privilegios suficientes. Esto permite implementar una capa de software en la base del micronúcleo que forme una interfaz con el hardware más especializada a la que se suele llamar *capa de abstracción del software*. Esta capa sirve para ocultar los detalles específicos del hardware de cada sistema, y es muy útil para portar el S.O. a distintas plataformas: una vez programada se puede obtener un nuevo S.O. simplemente recompilando el resto de programas.

Ahora es mucho más fácil insertar nuevos sistemas de archivo, o cambiar de gestor de memoria, ya que la interfaz ha quedado muy simplificada, y los módulos no son tan dependientes entre sí, ni están tan atados al hardware. Si uno de estos módulos fallara, no afecta

significativamente al resto de módulos siempre y cuando se pongan los mecanismos necesarios para que se recuperen de los errores.

Los sistemas micronúcleo han de ser diseñados con sumo cuidado, para proporcionar eficientes mecanismos de comunicación y sincronización entre tareas del sistema. Se pretende contrarrestar la pérdida de rendimiento respecto al modelo monolítico en el que las rutinas del núcleo se intercambiaban datos directamente en las propias llamadas a función o poniéndolos en la memoria común, y no era necesario sincronizarlas. Pese a esta mínima caída de rendimiento estos sistemas son más seguros de utilizar. Al igual que con el resto de sistemas, se ofrece al programador de aplicaciones toda una biblioteca de funciones de bajo nivel, que oculta la interfaz con cada uno de los módulos.

Estos procesos son interrumpibles por el micronúcleo y en ocasiones mantienen partes mínimas de código ejecutándose en modo núcleo.

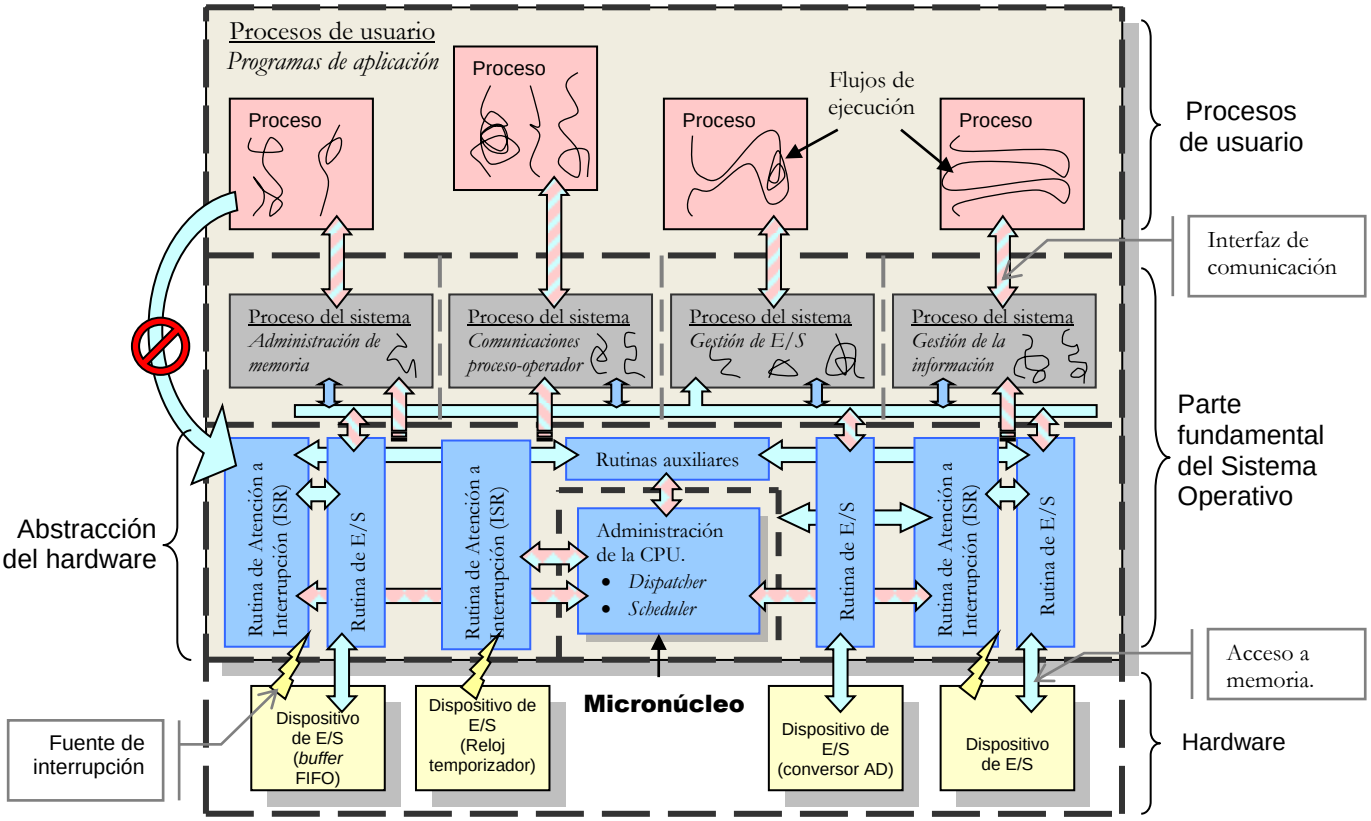


Figura 2-11. Sistema Operativo basado en micronúcleo

La **Figura 2-11** muestra una representación alternativa del un sistema basado en micronúcleo. En la capa inferior aparece el hardware del sistema. En la capa más interna aparece la gestión de la CPU encargada principalmente de asignar tiempos de procesamiento. La siguiente capa es de acceso a la E/S, y rutinas de atención a interrupción, encargadas del intercambio de información a más bajo nivel con el exterior.

La capa donde aparecen los procesos del sistema es donde se realiza el procesamiento principal para la administración de los recursos del sistema.

La capa más superior corresponde a los procesos de usuario, y es donde se ejecutarán las aplicaciones.

Las líneas zigzagueantes que aparecen representan los diferentes flujos de ejecución de los procesos.

En la práctica no se suelen implementar sistemas totalmente micronúcleo ni totalmente monolíticos, sino que se toman elementos de ambos, por lo que finalmente resulta que los grandes sistemas operativos suelen ser realmente híbridos. No obstante, por poner dos ejemplos habituales, decimos que **Linux** es un S.O. del estilo monolítico, mientras que **Windows XP** es de tipo microkernel, aunque ambos comparten elementos de ambos mundos.

2.3.1.5 Núcleo no apropiativo *(Non-Preemptive Kernel)*

El *núcleo no apropiativo* se llama también *núcleo cooperativo* debido a que las tareas sólo adquieren el control cuando lo necesitan en **coordinación** con otras tareas y eventos. Este tipo de núcleos podrían implementarse en aquellos sistemas que disponen de interrupciones por temporización. La conmutación entre tareas es muy rápida, ya que el mecanismo empleado es sencillo. Es usado en sistemas simples. Ejemplo: **MS-DOS**, **Windows 1.0 ~ 3.1**.

2.3.1.6 Núcleo apropiativo *(Preemptive Kernel)*

En el núcleo apropiativo, una tarea en ejecución puede ser desalojada por una tarea de mayor prioridad cuando esta pase a estar preparada, o por otra de igual prioridad cuando haya pasado una determinada cantidad de tiempo. El núcleo apropiativo se basa principalmente en el uso de las interrupciones del sistema, más concretamente las que generan los temporizadores.

Se debe disponer entonces de una interrupción de reloj para hacer un núcleo apropiativo real. El corazón de este tipo de núcleos es el cambio de contexto. Para su implementación, se debe poder manipular la pila, ya que estos núcleos han de ser capaces de conmutar entre varias.

2.3.1.7 Apropiativo Vs. No apropiativo.

El núcleo apropiativo:

- Es el más difícil de desarrollar, pero es más fácil de usar, aunque en ocasiones se usa incorrectamente.
- La determinación teórica de la ejecutabilidad de un sistema complejo es más complicada que con un núcleo no apropiativo.
- Es más complejo, pero maximiza el uso de la CPU al permitir la coexistencia de tareas de TR y otras tareas no críticas.

El núcleo no apropiativo:

- A menudo se debe invertir más tiempo programando el resto del sistema debido a las capacidades tan limitadas del núcleo, pero es mejor para aquellos procesadores que estén técnicamente muy limitados como los que no disponen de interrupciones por temporización.
- Puede ser más predecible en los tiempos de respuesta para STR simples, pero en esos casos se requiere un conocimiento temporal exacto de todas las tareas que intervengan.
- Es más fácil calcular teóricamente el máximo tiempo en completar un trabajo determinado, ya que no será interrumpido por otros, y la sincronización entre tareas está implícita en el código.

2.4 Elementos de un SOTR

Cualquier Sistema Operativo de Tiempo Real tiene elementos en común con los SS.OO. genéricos. No obstante, hacen hincapié en las cuestiones relacionadas con la interacción con el mundo exterior, la fiabilidad y la predecibilidad.

2.4.1 Uso de POSIX en los SOTR

La normativa **POSIX** IEEE 1003 (**P**ortable **O**perating **S**ystem **I**nterface, Uni**X** based) surgió ante la necesidad de normalizar ciertos aspectos de los sistemas operativos, como son las llamadas a biblioteca para solicitar los servicios del sistema. Uno de los objetivos es lograr la portabilidad de las aplicaciones a nivel de código fuente. Pese a que esta especificación originalmente se creó enfocada hacia los sistemas operativos estilo **UNIX**, los SOTR se pueden ver igualmente beneficiados usando solamente un subconjunto de funciones **POSIX**. Las funciones de mayor interés son las encargadas de la creación y destrucción de tareas, su gestión, el manejo de objetos de sincronización, y el control del tiempo. La normalización de algunas de estas funciones es relativamente moderna, por ello existe una gran cantidad de SS.OO. que no las soportan completamente. El caso de **Windows** es un claro ejemplo: fue diseñado años antes de que **POSIX** definiera las rutinas de gestión de hilos, por lo que cuenta con sus propios mecanismos, más acordes con la filosofía del sistema.

Los pequeños SOTR a menudo no hacen uso de normativa alguna en el diseño de las funciones para el programador, pero aquellos que siguen algún estándar suelen adoptar **POSIX**. Ejemplo de estos sistemas son **vxWorks**, **RTLinux 3.x**, y **RTAI** (mediante bibliotecas adicionales).

Otros estándares que se usarán son los propuestos por **The Open Group Base Specifications**, que define la **Single UNIX Specification**, **IEEE 1003**, las recomendaciones del lenguaje **c99**, y de forma más global la normalización **ISO**. A menudo ocurre que podemos contar con un mecanismo que no está recogido en ninguna normalización, pero se encuentra omnipresente en distintas plataformas; tal es el caso de las herramientas del proyecto **GNU**, que a menudo aporta más funcionalidad que la que dictan los estándares. El código aquí presente asume que se está trabajando con **GNU**.

2.4.2 Concurrencia

La *concurrencia* es la capacidad de un sistema de realizar varias actividades en paralelo. La concurrencia para el núcleo del S.O., implica que se han de mantener varias tareas en estados intermedios de su ejecución. El sistema puede alterar dichos estados dependiendo del desarrollo de cada uno de los programas, y de otras decisiones internas. Cada tarea por separado pretenderá hacer uso de los recursos del sistema, para lo cual deben estar correctamente asignados.

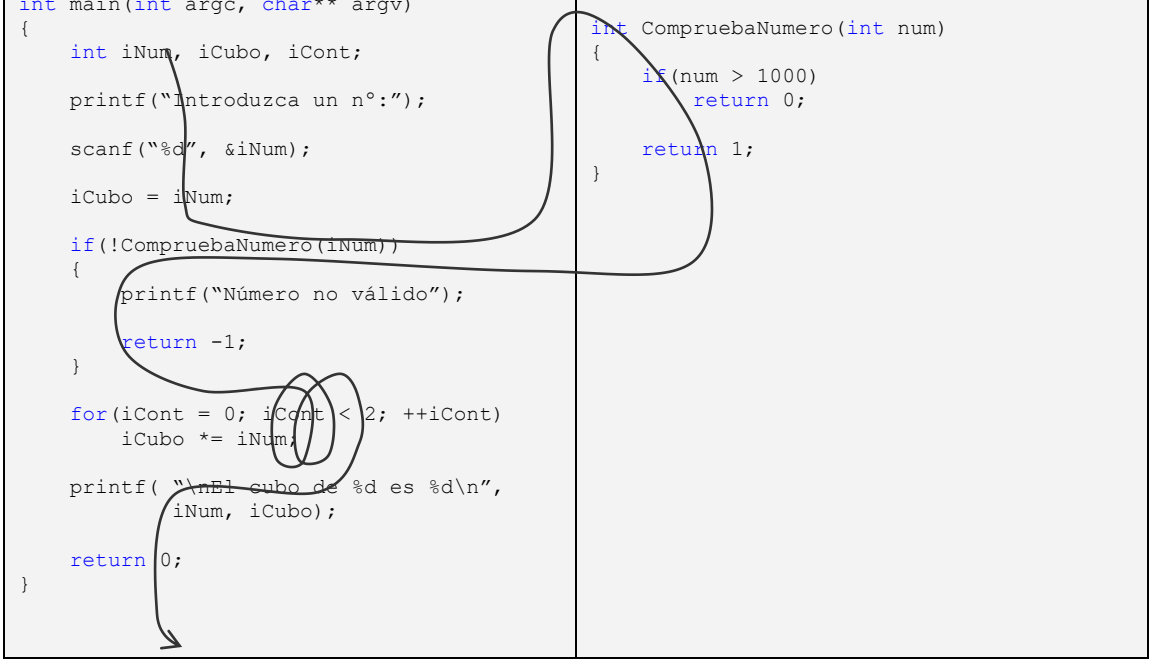
Si dos o más tareas pretenden usan simultáneamente un mismo recurso por ejemplo el código máquina de una subrutina, se dice que lo usan concurrentemente. La concurrencia se da en los sistemas de tiempo compartido, sistemas multiprocesadores, y los sistemas distribuidos.

La concurrencia en estos términos apareció cuando se pretendía maximizar el uso de CPU entre diferentes aplicaciones, sin malgastar los tiempos de espera en el acceso a dispositivos lentos. Así surgió la idea de sistema de tiempo compartido, en los que se asignaba una cantidad de tiempo a cada trabajo para ser procesado, y llegándose a ciertos momentos, se conmutaba de un trabajo a otro para que todos fueran siendo ejecutados poco a poco.

Código en lenguaje C

```
#include <stdio.h>

int CompruebaNumero(int num);

int main(int argc, char** argv)
{
    int iNum, iCubo, iCont;

    printf("Introduzca un n°:");

    scanf("%d", &iNum);

    iCubo = iNum;

    if(!CompruebaNumero(iNum))
    {
        printf("Número no válido");

        return -1;
    }

    for(iCont = 0; iCont < 2; ++iCont)
        iCubo *= iNum;

    printf( "\nEl cubo de %d es %d\n",
            iNum, iCubo);

    return 0;
}
```

```
int CompruebaNumero(int num)
{
    if(num > 1000)
        return 0;

    return 1;
}
```



Programa 2-2. Flujo de ejecución de un programa.

En el **Programa 2-2** se ha dibujado una línea que marca el flujo de ejecución de una tarea. Al igual que el flujo pasa por la función `CompruebaNumero(…)` que es invocada dentro de `main(…)`; también entraría a `printf(…)` y `scanf(…)`, aunque no se hayan representado.

El **Programa 2-3** incorpora el código de una rutina ejecutada en respuesta a una interrupción (ISR) de entrada de datos serie. Por simplicidad, asumiremos que esta ha sido configurada antes de entrar al `main()`. El atributo que se le ha añadido **`__attribute__((interrupt))`** es propio del compilador **gcc** (no es estándar) y significa que debe generar el código correspondiente a una ISR. Por ejemplo en el 68HC11 indicaría que se use la instrucción `rti` en lugar de `rts` para regresar de la interrupción, y no se usen argumentos de función ni retorno de datos..

Obsérvese cómo ahora aparecen dos líneas: una sigue el flujo de ejecución del programa principal, y la otra el de la ISR. Ésta puede interrumpir la ejecución del hilo principal en el momento que marque el temporizador que la activa, para ejecutar su propio código. Decimos entonces que la ejecución es *concurrente*. Una vez que termine `timer_interrupt()`, seguiría ejecutándose normalmente el código principal. En este caso ambos programas comparten la ejecución de la función `CompruebaNumero()`; esto la califica como un recurso de código compartido. Existe cierta probabilidad de que la interrupción llegue justo cuando el hilo principal se encontraba ejecutándola.

Código en lenguaje C

```
#include <stdio.h>

int CompruebaNumero(int num);

int main(int argc, char** argv)
{
    int iNum, iCubo, iCont;

    printf("Introduzca un n°:");

    scanf("%d", &iNum);

    iCubo = iNum;

    if(!CompruebaNumero(iNum))
    {
        printf("Número no válido");

        return -1;
    }

    for(iCont = 0; iCont < 2; ++iCont)
        iCubo *= iNum;

    printf( "\nEl cubo de %d es %d\n",
            iNum, iCubo);

    return 0;
}
```

```
int CompruebaNumero(int num)
{
    if(num > 1000)
        return 0;

    return 1;
}


void __attribute__ ((interrupt))
timer_interrupt (void)
{
    int num = GetVal();

    if(!CompruebaNumero(num))
        return;

    PutVal(num >> 2);
}
```

Programa 2-3. Ejecución de dos rutinas en paralelo con código compartido

2.4.2.1 Reentrada *(Reentrancy)*

En un sistema informático, dos o más tareas podrían querer usar la misma subrutina, como ocurre con la función `CompruebaNumero()`. El problema de las interrupciones hardware es que no se puede saber cuándo se retirará a una tarea de la ejecución, y cuándo ocupará su lugar otra tarea. Si desde dentro del código común se hace uso de algún recurso compartido por dichas tareas como pueda ser la posición de memoria que ocupa una variable global de C, o un dispositivo hardware, podrían ocurrir corrupción de datos si no se realiza correctamente. El problema vendría a raíz de las operaciones de lectura–modificación–escritura que hicieran las tareas sobre la información del recurso. Si este proceso es demasiado lento, cabría la posibilidad de que las tareas se interrumpieran las unas a las otras a la mitad del proceso, siendo erróneas las escrituras de datos.

Al **Programa 2-4** se le ha añadido la variable global `g_num`[7] que acumula el valor `num` que se le envíe como argumento a la función `CompruebaNumero()`. Para realizar este incremento, la CPU ha de llevar el valor actual de `g_num` a un registro interno, sumarle el valor de `num`, y volverlo a escribir en memoria. Si mientras que se estaba procesando la llamada que se realiza desde `main()` apareciera la interrupción, cabría la posibilidad de haber interrumpido el proceso de acumulación justo antes de actualizar `g_num` con el valor recién incrementado. La ISR podría igualmente actualizar el valor de `g_num`, pero al volver de la interrupción ese valor será sobrescrito con el que se había calculado en la rutina principal. Como resultado, la acumulación hecha desde la interrupción no ha tenido efecto, al haber sido machacada por el valor calculado por la llamada que se efectuó desde `main()`.

[7] La declaración de la variable `g_num` lleva el modificador `volatile`, que es usado para indicarle al compilador que esa variable va a ser accedida por varias tareas concurrentemente. De esta forma el compilador no optimizará el acceso a la misma, para evitar que termine existiendo únicamente en un registro de la CPU.

Código en lenguaje C

```
#include <stdio.h>

volatile int g_num;

int CompruebaNumero(int num);

int main()
{
    int iNum, iCubo, iCont;

    printf("Introduzca un n°:");
    scanf("%d", &iNum);

    iCubo = iNum;
    if(!CompruebaNumero(iNum))
    {
        printf("Número no válido");

        return -1;
    }

    for(iCont = 0; iCont < 2; ++iCont)
        iCubo *= iNum;

    printf( "\nEl cubo de %d es %d\n",
            iNum, iCubo);

    return g num;
}
```

```
int CompruebaNumero(int num)
{
    if(num > 10)
    {
        g num = g num + num;

        return 0;
    }

    return 1;
}




void   attribute  ((interrupt))
timer interrupt (void)
{
    int num = GetVal();

    if(!CompruebaNumero(num))
        return;

    PutVal(num >> 2);
}
```

Programa 2-4. Modificación de una variable global compartida.

Si una rutina usara sólo variables locales o pasadas como argumentos, estaría libre de errores de este tipo, al almacenarse estas en la pila de cada tarea, como ocurre con la función `CompruebaNumero()`. Decimos entonces que es una *rutina reentrante*, o simplemente que es una *rutina pura*. En una rutina pura no se debe hacer uso de variables globales o de dispositivos *hardware*, ya que son susceptibles de ser usados por varias tareas. Una manera de permitir que varias tareas utilicen con seguridad un recurso compartido, es convertir al acceso en indivisible para impedir que cuando una tarea esté utilizándolo, puedan hacerlo las demás. El acceso indivisible también se conoce como acceso *atómico*. Esto podría conseguirse si se contara con los privilegios necesarios, deshabilitando las interrupciones durante el tiempo de acceso al recurso común. Por razones de seguridad, lamentablemente no sería posible si estamos en modo usuario. Otra forma adicional es contar con instrucciones que ya de por sí se ejecuten indivisiblemente[8], muy habituales en los micros CISC, aunque esta solución sólo es válida si la modificación de datos es tan trivial como para que una sola instrucción la pueda hacer.

[8] En los micros modernos pueden existir ciertas interrupciones como la de fallo de página que hagan que interrumpan incluso la ejecución de una instrucción del micro.

Ejercicios

14. Realizar las modificaciones oportunas al Programa 2-4 para ejecutar la función `CompruebaNumero()` de la forma más segura. Para ello se cuenta con las llamadas `HabilitaInt()` y `DeshabilitaInt()` que respectivamente habilitan y deshabilitan las interrupciones enmascarables.

```
#include <stdio.h>

volatile int g_num;
int CompruebaNumero(int num);

int main()
{
    int iNum, iCubo, iCont, iRes;

    printf("Introduzca un n°:");
    scanf("%d", &iNum);

    iCubo = iNum;

    DeshabilitaInt();
    iRes = CompruebaNumero(iNum);
    HabilitaInt();

    if(!iRes) {
        printf("Número no válido");

        return -1;
    }

    for(iCont = 0; iCont < 2; ++iCont)
        iCubo *= iNum;

    printf( "\nEl cubo de %d es %d\n",
            iNum, iCubo);

    return g_num;
}

int CompruebaNumero(int num)
{
    if(num > 10) {
        g_num = g_num + num;

        return 0;
    }

    return 1;
}

void __attribute__((interrupt))
timer_interrupt (void)
{
    int num = GetVal();

    if(!CompruebaNumero(num))
        return;

    PutVal(num >> 2);
}
```

La solución ha consistido en hacer ininterrumpible la llamada a la función `CompruebaNumero()` que se hace desde `main()`. Ahora el proceso de lectura–modificación–escritura de `g_num` se realiza de forma atómica por lo que no hay riesgo de obtener resultados erróneos.

La deshabilitación de interrupciones implica que la latencia de interrupción en el peor de los casos será mayor, por lo que hay que usarla con suma cautela.

Observe que en este caso no se podría realizar esta deshabilitación dentro de la función `CompruebaNumero()` aunque con ello se colabore a restringir el código protegido:

```
int CompruebaNumero(int num)
{
    if(num > 10)
    {
        DeshabilitaInt();
        g_num = g_num + num;
        HabilitaInt();

        return 0;
    }

    return 1;
}
```

En este último caso la llamada se realizara desde la ISR, podrían habilitarse las interrupciones antes de salir del manejador pudiendo aparecer efectos colaterales perjudiciales.

2.4.2.2 Multiprogramación

La *multiprogramación* es una evolución de los primeros sistemas de tiempo compartido en los que se transmite la impresión de varias tareas trabajando al mismo tiempo. Cada tarea cree que posee a la CPU, pero esta ilusión es creada por el sistema, que realiza una rápida conmutación entre ellas. Sólo una tarea puede estar usando la CPU en un momento dado. Quedará trabajo sin acabar que será realizado por otras tareas que están a la espera del uso de la CPU. En estos casos se habla de *seudoparalelismo*, o *paralelismo a nivel se software*, en comparación

con el paralelismo real del hardware que consiste en el trabajo simultáneo de la CPU y los dispositivos de E/S, o de varios procesadores al mismo tiempo. La multiprogramación engloba la gestión de interrupciones, eventos, comunicación, compartimiento de datos, y la temporización. Las ventajas fundamentales de la multiprogramación son:

- Se aprovecha la CPU durante los tiempos de inactividad de un proceso para que vayan ejecutándose otros en el intervalo.
- Se van consiguiendo resultados en paralelo, por lo que los tiempos de 1ª respuesta son relativamente bajos, y adecuados para compartir interactivamente la máquina.

2.4.2.3 Multiproceso

El *multiproceso* o *multiprocesamiento* es una característica de muchos sistemas que indica que existe una ejecución real en paralelo de diferentes tareas cada una corriendo simultáneamente en un procesador diferente. Suele ocurrir que para maximizar el uso del hardware un sistema con multiproceso emplee a su vez multiprogramación en cada uno de los procesadores que incorpora.

Se llamaría *multiproceso simétrico* (SMP) si todos los procesadores son de la misma naturaleza y pueden ejecutar indistintamente cualquiera de las tareas encomendadas. En caso de que cada procesador se especialice en un único tipo de procesamiento, se denomina *multiproceso asimétrico*.

Al procesador principal de un sistema multiprocesador se le denomina Unidad Central de Procesamiento (CPU)[9].

2.4.3 Procesos

Las técnicas de diseño de Tiempo Real permiten al diseñador partir problemas grandes y complejos en tareas más simples. Esas unidades de código (programas) más manejables permiten una respuesta más rápida para los eventos importantes, a la vez que priorizan los trabajos a realizar, en un formato estructurado de efectividad comprobada. Llamamos *proceso* a cada una de estos programas en ejecución. El núcleo realiza el trabajo de administrar el tiempo, sincronizar, y mantener una comunicación fluida entre ellos. En la misma cantidad de tiempo ahora pueden ser realizadas más actividades, permitiendo a unas tareas trabajar mientras que otras esperan a que se produzcan ciertos eventos.

Los procesos son los mecanismos de que dispone el S.O. para ejecutar tareas de la forma más segura y eficiente posible. A menudo en la literatura se encuentran como sinónimos los términos tarea y proceso[10], por ser los procesos los encargados de ejecutar las tareas.

El concepto de proceso ha evolucionado notablemente desde su definición original. En la actualidad, hemos de concebir a los procesos más bien como meros contenedores de recursos entre los que podemos encontrar: tareas[11] encargadas de realizar la ejecución del

[9] Muchos autores llaman CPU a cualquier procesador programable. Nosotros sólo consideraremos CPU al procesador principal de un computador.

[10] En general no vamos a considerar esa equivalencia, ya que tarea es un concepto que existía desde antes de la aparición de los procesos, pudiendo ser utilizado incluso en el caso de sistemas que no incorporen S.O., lo que es habitual en computadores muy pequeños. Consideraremos como tarea incluso a una rutina de atención a interrupción (RAI) que esté en ejecución.

[11] Originalmente los procesos sólo mantenían una tarea en ejecución; de ahí que la equivalencia entre tarea y proceso tuviera más sentido en la literatura antigua. En la actualidad pueden agruparse varias tareas que comparten recursos y sistemas de protección, llamándoselas en tales casos hilos.

código, memoria para almacenar entre otras cosas el estado de las tareas, descriptores[12] de archivos abiertos, y otros objetos del sistema que indican qué elementos hardware o software tienen en usufructo. Cuando se destruye un proceso, el sistema se encarga de liberar automáticamente todos los recursos que tuviera reservados en exclusividad.

Es posible tener en un mismo proceso tareas de tiempo real coexistiendo con otras tareas de requerimientos temporales más relajados, siendo función del S.O. el asegurar que las primeras puedan generar sus resultados con eficacia en los tiempos exigidos. El comportamiento de una tarea de TR debe ser independiente de la velocidad de procesamiento, de la carga del sistema, y del entorno donde se encuentre, de tal forma que sea reproducible bajo distintas condiciones.

Mediante la biblioteca GNU C, podemos crear nuevos procesos mediante la función `system()`, cuyo único argumento es una cadena de texto que será pasada al intérprete de comandos, lo que equivale a haberla introducido por la línea de órdenes.

```
//
// Crar un proyecto para ejecutar un programa que está el en disco
//
#include <stdlib.h>

int main ()
{
    // Ejecuta la orden del sistema
    //    ls -l
    // en su propio proceso.
    system("ls -l");

    // Otras cosas que hacer

    return 0;
}
```

Programa 2-5. Creación de un nuevo proceso.

En el ejemplo se llama al programa `ls` con el modificador `-l` para que saque un listado ancho del directorio actual por pantalla. Al igual que otras llamadas POSIX, el entero retornado por `system()` será -1 si por alguna razón la función falla[13], y será mayor o igual a 0 si funciona correctamente. Este será el comportamiento habitual en el resto de funciones POSIX que utilicemos, si no se indicara lo contrario.

Los procesos son identificados unívocamente por el sistema mediante un número de tipo `pid_t`. Cualquier proceso puede obtener su propio identificador llamando a la función `getpid()`. En POSIX existe una relación jerárquica entre los procesos. Al proceso que crea a otro se le llama *proceso padre*, y al creado *proceso hijo*. Esta relación de parentesco no tiene por qué existir en los sistemas no POSIX. La creación de un proceso se llama *nacimiento*, y su destrucción *muerte*.

El método clásico para crear un nuevo proceso es mediante la llamada al sistema `fork()`[14] que se utiliza para clonar al que la llame. Una vez ejecutada la función, en el proceso padre retornará el identificador del proceso hijo, y en el hijo retornará 0.

[12] Los descriptores son identificadores con los que las tareas pueden referenciar a los archivos. También son llamados manejadores.

[13] El fallo que se devuelve en el código de retorno de las funciones no se refiere al fallo del que se habló en los STR del tema 1. A lo que se refiere es que la función no ha podido cumplir su objetivo, notificándolo de esta manera. Un fallo informático en el sentido estricto no serían notificados porque no se habrían tenido en cuenta durante el desarrollo.

[14] La función `fork()` es realmente una interfaz de la llamada al sistema correspondiente; `clone()` en el caso de Linux. Realiza una clonación parcial del proceso que la llame.

```
//
// Programa que crea un proceso que es una copia de sí mismo
//
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main ()
{
    pid_t child_pid; // Identificador de proceso

    printf ("El identificador del programa principal es %d\n", (int) getpid ());

    child pid = fork (); // Realiza la clonación del proceso

    // En cada proceso la función fork devuelve distintos identificadores
    if (child pid != 0)
    {
        // Este bloque sólo será ejecutado por el padre

        printf ("Este es el proceso padre, con identificador %d\n"
                "El identificador del proceso hijo es %d\n",
                (int) getpid (), (int) child pid);
    }
    else
    {
        // Este bloque sólo será ejecutado por el hijo

        printf ("Este es el proceso hijo, con identificador %d\n", (int) getpid ());
    }

    // Todo lo que se ejecutara aquí se procesaría por duplicado:
    //  tanto por el proceso padre como por el hijo

    return 0;
}
```

Programa 2-6. Clonación de un proceso

Es posible que quisiéramos detener la ejecución de un hilo hasta que termine el proceso hijo. Para ello podemos usar las funciones `wait()` y `waitpid()`. Las dos dejan suspendido al hilo que las invoca hasta que termina algún proceso hijo del que se obtiene su código de retorno. Si en el momento de llamarlas ya hubiera alguno que hubiera terminado, se regresa inmediatamente sin detener la ejecución.

```
#include <sys/wait.h>

pid_t wait (int *stat_loc);
pid_t waitpid (pid_t pid, int *stat_loc, int options);
```

- **pid**: identificador del proceso por el que se está esperando. Usando -1, se espera por cualquier proceso hijo. Cualquier otro valor negativo del pid hace referencia al grupo del proceso en lugar de su identificador.
- **stat_loc**: dirección de un int que será rellenado con el estado devuelto por el `return` de la función `main()`, o por el código de `exit()` del proceso. Pasando NULL, no se devuelve estado alguno.
- **options**: Opciones combinadas por el operador |: `WNOHANG` indica que no suspenda al proceso, `WUNTRACED` indica que notifique a los procesos detenidos (STOP) además de los terminados. Usando 0 se obtendrá el comportamiento estándar.

En el caso de usar un `pid` negativo, si varios procesos hubieran terminado, se devuelve la información de uno de ellos elegido al azar, siendo necesario volver a llamar a la función para recuperar el estado del resto.

La llamada `wait (&codigo_ret)` equivale a `waitpid (-1, &codigo_ret, 0)`.

Ahora podríamos insertar el siguiente código al **Programa 2-6** para esperar a que termine el proceso recién creado:

```
    //
    // El código que se coloque aquí se ejecutará en paralelo con el del hijo
```

```
    // . . .

    wait(NULL); // Espera a que termine cualquier hijo

    // El código que se coloque aquí se ejecutará una vez que el hijo haya
    //  concluido.
    // . . .

    return 0;
}
```

Programa 2-7. Espera a la terminación de un proceso hijo

La forma habitual en que termina un proceso es saliendo de la función `main()` lo que indica al sistema que puede liberar los recursos que hubiera utilizado. También es posible terminar un proceso mediante una llamada explícita a la función `exit()`, que toma como único argumento el entero que devolverá el proceso en lugar del que se hubiera devuelto por el `return` de `main()`. La función `exit()` realiza algunos trabajos de limpieza previos a la terminación del proceso, como el vaciamiento de buffers antes de cerrar los archivos que quedaran abiertos. Si no quisiéramos ese comportamiento adicional, usaremos `_exit()` en su lugar.

En el siguiente programa, se utiliza la función `malloc()` para hacer una petición de memoria con capacidad para 500 números. Si no pudiera alojarlos, el sistema devolvería NULL, y en tal caso interrumpiremos la ejecución del proceso:

```
/*
 *  Liberación automática de memoria, y salida prematura de un proceso.
 */
#include <memory.h>

int main()
{
    short *pMem;

    pMem = (short*) malloc(500 * sizeof(*pMem));

    if(pMem == NULL)
        exit(-1);

    // Utilizar la memoria reservada a la que apunta pMem
    // . . .

    return 0;
}
```

Programa 2-8. Muerte de un proceso

En este programa de ejemplo en ningún momento libera la memoria explícitamente mediante `free()` u otra función similar, por lo que en el instante previo a terminar, tendríamos una laguna de memoria, es decir, una reserva de memoria que no es referenciada. El S.O. es el encargado de liberar esa memoria durante la muerte del proceso, en su labor de seguimiento y limpieza de los recursos que se hubieran reservado. No obstante, aunque el sistema lo permita, el no liberar explícitamente los recursos que se hayan reservado, se considera una mala práctica de programación.

2.4.3.1 Cambio de Contexto / Conmutación entre tareas *(Context Switch / Task Switch)*

El núcleo es el encargado de decidir qué tarea debe ser ejecutada. Para ello cuenta con un sistema de prioridades que le sirve para decidir cuál es la siguiente tarea candidata a usar la CPU, y con un sistema de permisos para el control del resto de recursos. Por ejemplo las tareas

más críticas para el buen funcionamiento del sistema serán las más prioritarias[15], pudiendo hacer uso de la CPU cuando la necesiten, interfiriendo si acaso a otras tareas menos prioritarias. Se dice en tal caso que la tarea más prioritaria ha interferido a la menos prioritaria.

Cuando una tarea se impone sobre otra ya sea porque la bloquee o porque la interfiera, se salvan los registros de la CPU, para ser remplazados por los de la tarea entrante que a su vez habían sido salvados con anterioridad. La tarea que adquiere la CPU continúa por tanto donde lo había dejado, o bien comienza su ejecución si es que acaba de ser creada.

En el caso de contar con tareas soportadas por el sistema operativo en forma de procesos, además de los registros de la CPU, también se han de actualizar todas las referencias a sus estructuras de datos características, principalmente el mapa de memoria al que accede. A los datos particulares que necesita un proceso para poder ejecutarse le llamamos contexto. La conmutación de un contexto a otro entre diferentes procesos lo llamamos *cambio de contexto*. Los procesos que administra el núcleo tienen un área de almacenamiento de cambio de contexto (*context switch storage area*), en donde se almacena datos importantes relativos a los recursos que mantiene como son los registros de la CPU, referencias al mapa de memoria, a los archivos, y a los dispositivos que están siendo usados. La rapidez del cambio de contexto, influye significativamente en la eficiencia del sistema.

El cambio de contexto se desencadena habitualmente por ciertas interrupciones:

Interrupción software provocada por una llamada al sistema. Esto es lo habitual en aplicaciones de E/S intensiva, ya que hacen muchas llamadas que conllevan respuestas diferidas como el acceso a datos en dispositivos de almacenamiento masivo, dándole pie al núcleo para conmutar a otros hilos.

Interrupción hardware generada por un temporizador preparado por el núcleo para una posible conmutación forzosa por si acaso la tarea no lo hiciera voluntariamente. Esto es lo habitual en aplicaciones de cómputo intensivo.

En ocasiones, si al hablar del cambio de contexto se quiere dar especial énfasis al hecho de que lo que se está cambiando es de tarea, se utiliza el término *conmutación de tareas*, que hace referencia únicamente a las que son mantenidas por el núcleo.

El programa encargado de realizar la conmutación de una tarea a otra se llama *despachador* (*dispatcher*) cuya función principal es sacar a la que estaba anteriormente en ejecución, y asignar la CPU a la nueva. Actualizará las estructuras de datos que se vayan a necesitar incluido, el contexto del proceso, aunque su cometido está muy cercano al hardware de la CPU y por tanto emplea necesariamente ensamblador.

El programa encargado de elegir a la siguiente tarea en utilizar la CPU se llama *planificador* (*scheduler*) y utiliza diferentes algoritmos para determinar el siguiente hilo al que se le debe asignar la CPU. Muy frecuentemente la algorítmica usada emplea técnicas estadísticas y heurísticas para tomar las decisiones más correctas. Hay muy distintas razones que pueden justificar la elección de la siguiente tarea. Suelen estar íntimamente relacionadas con la prioridad que el programador les dé, la cantidad de E/S que realicen, o la carga actual del sistema. En el siguiente tema se citarán algunas razones que el programador ha de tener en cuenta a la hora de implementar o elegir un STR.

El despachador junto con el planificador forman el corazón de cualquier S.O. moderno. Son los únicos programas que se han de implementan obligatoriamente en los micronúcleos.

[15] En este contexto, las tareas más prioritarias serán normalmente las ISR, seguidas por las rutinas del núcleo (*kernel tasks*), las tareas de usuario de tiempo real, y finalmente las tareas de usuario normales. Dentro de cada uno de estos grupos existe además sistemas de prioridades específicos.

En la **Figura 2-12** se muestra un sistema de varias tareas en ejecución. Es objetivo del núcleo es que utilicen la CPU de la forma más eficiente. En la parte de la izquierda aparecen varias bandas representando tareas dispuestas a ejecutarse. Las franjas oscuras representan los tiempos de procesamiento que necesita cada tarea, mientras que los espacios intermedios que quedan en blanco representan intervalos de espera necesarios bien para acceder a dispositivos de E/S o a que finalice una temporización. En todos los casos se pretende mostrar la forma en que cada tarea esperaría ser ejecutada si sólo fuera la única en el sistema. Sin embargo, el tiempo de CPU debe ser repartido entre todas. Para ello a manera de multiplexor el planificador selecciona una tarea candidata, que entrega al despachador. Cada cierto tiempo, el despachador interrumpirá la tarea que estuviera haciendo uso de la CPU, para sustituirla por la que haya propuesto el planificador.

Las dos bandas superiores representan tareas de tiempo real, en las que podemos apreciar su carácter de ejecución periódica. Las tareas de TR hacen uso de temporizadores para indicar al sistema cuándo deben volver a ejecutarse.

Las dos bandas superiores representan tareas de tiempo real, en las que podemos apreciar su carácter de ejecución periódica. Las tareas de TR hacen uso de temporizadores para indicar al sistema cuándo deben volver a ejecutarse.

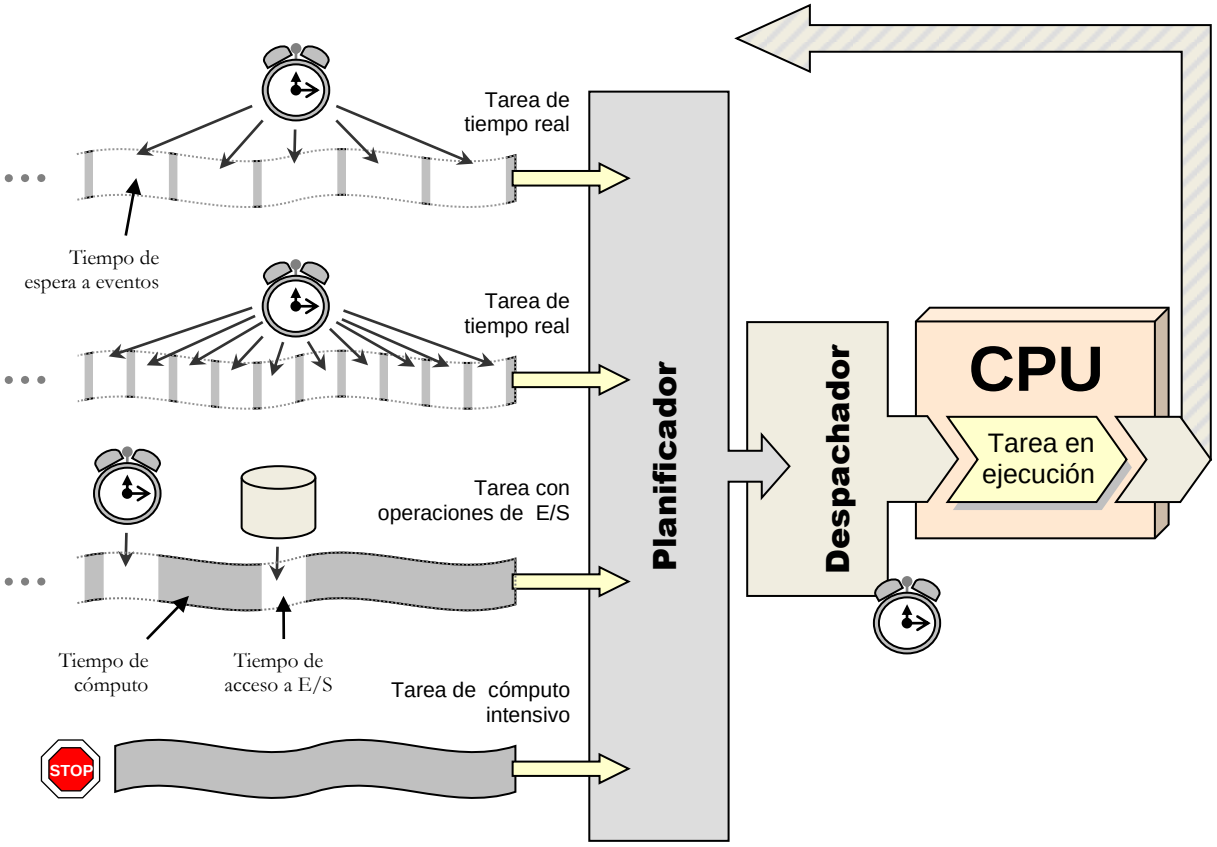


Figura 2-12. Colaboración entre planificador y despachador.

Las dos bandas inferiores representan tareas de procesamiento interactivo. La superior es interrumpida por acceder a un dispositivo lento, en este caso el disco duro, y por peticiones explícitas de espera temporizada; todas ellas originadas por una llamada al sistema. Este es un ejemplo de la típica tarea usada para interactuar con los usuarios, u operarios del sistema.

La continuidad de la banda inferior indica que se trata de una tarea que usaría intensivamente la CPU durante todo su tiempo de vida, posiblemente para hacer cómputos matemáticos. En ningún momento dará pie a ser interrumpida, al menos por su propia iniciativa. En este caso la tarea tiene un final bien delimitado, que se alcanza cuando haya terminado su procesamiento.

2.4.4 Hilos (Threads)

Hasta ahora hemos visto que los procesos son los encargados de realizan trabajos. Estructurar un problema en varios procesos puede incrementar el rendimiento de la aplicación y disminuir el tiempo de respuesta al permitir la ejecución de código asincrónicamente y paralelo. Sucede muy a menudo que desearíamos que estos procesos pudieran compartir recursos libremente entre ellos para simplificar el intercambio de datos. La solución estaría en permitir una ejecución paralela de las diferentes partes del programa dentro del proceso. A estas unidades de ejecución se les llama frecuentemente *subprocesos* o *procesos ligeros*, por su similitud con los procesos clásicos que ahora pasan a denominarse *procesos pesados*.

En la actualidad se emplea otra denominación más moderna que llama a estas tareas *hilos*. El proceso es un mero contenedor de los recursos que utilice la aplicación, entre los que cabe destacar: la memoria que tenga asignada en la que se almacenan las variables, los objetos del sistema que se manejen[16], archivos actualmente abiertos, temporizadores,… y por supuesto los hilos de los que haga uso. El término hilo se debe al aspecto que tienen las líneas que siguen el flujo de los programas en ejecución tal como se mostraba en el **Programa 2-2**.

Lógicamente cada proceso tendrá por lo menos un hilo. Los hilos son realmente entes de ejecución de código, tareas que comparten recursos. Un hilo mantiene un estado definido principalmente por su propia pila y los registros del procesador a nivel de usuario, y una estructura interna del núcleo con información adicional del estado. En la pila es donde se almacena la parte fundamental de su estado, que incluye principalmente a las variables locales (automáticas). El resto de memoria como la asignada mediante `malloc()`, o la reservada para variables globales, será compartida por todos los hilos del proceso. Es importante remarcar el hecho de que las variables locales de un hilo no se comparten con otros hilos, aunque son físicamente accesibles desde cualquier hilo dentro de un proceso. La consecuencia que un hilo mal programado podría corromper los datos de otros hilos sin que exista ningún mecanismo de protección incluso en las máquinas que sí los ofrecieran. Este es el principal argumento de los detractores de los hilos, que proponen que cada tarea sea ejecutada desde la seguridad de un proceso. Las ventajas que se citan a su favor son:

- La posibilidad de usar recursos compartidos libremente dentro del proceso, sobre todo cuando hayan tareas que necesiten el resultado de otras tareas. Éste no sería un dato a considerar si cada tarea utiliza sus propios recursos exclusivamente, y es independiente de las demás.
- Rapidez en la creación y destrucción de los hilos, ya que implican escasa sobrecarga al sistema. Únicamente es destacable la reserva de memoria necesaria para la nueva pila. Por otra parte no se crean y destruyen tareas tan a menudo como para que sea un factor decisivo.
- Ágil conmutación entre los hilos que están dentro de un mismo proceso. Aunque podrían optimizarse los SS.OO. para que la conmutación entre diferentes procesos no sea demasiado costosa.

[16] Es común en algunos SS.OO. llamar objetos a los recursos del sistema en general, como los relacionados con los gráficos (*bitmaps*, brochas, pinceles,…) , el sonido, la sincronización, etc.

- Aunque uno de los hilos de un proceso se encuentre bloqueado, por ejemplo a la espera de recibir datos del exterior, el resto de hilos puede seguir su ejecución normalmente. Esto es lo que los hace tan atractivos para ocuparse de labores de E/S asíncrona del proceso.
- Se podría dedicar ciertos hilos a hacer cálculos de larga duración sin tener que paralizar a todo el proceso.
- Se puede asignar prioridades a los hilos para jerarquizar las tareas del proceso en base a su importancia.

Pese a esto, en ocasiones puede ser preferible que dos o más procesos compartan únicamente el área de memoria estrictamente necesaria para comunicarse, con lo que se podrían beneficiar así de la protección de memoria entre diferentes procesos.

La capacidad que tiene un sistema para disponer de varios hilos por cada proceso se llama *multihilado*. Consiste en la división de un mismo proceso en diferentes flujos de ejecución que son ejecutados en paralelo (al menos en apariencia). Esto permite que las diferentes partes del proceso sean llevadas a cabo independientemente.

El multihilado permite conseguir un mayor aprovechamiento del sistema, logrando una utilización de los recursos de la máquina con un "grano más fino" que lo que se conseguía con procesos de una sola tarea. Además disminuye el tiempo medio de ejecución, acorta los plazos de primera respuesta, y permite la implementación de una comunicación más fluida con el mundo exterior. También aumenta la escalabilidad ya que en los sistemas multiprocesador se puede asignar los hilos para ser ejecutados en paralelo cada uno en un procesador distinto, con la ventaja de utilizar un mismo código binario tanto para sistemas basados en una sola CPU, como para sistemas multiprocesadores.

El hilo es de especial utilidad en los sistemas cuyos núcleos permitan la multitarea con expulsión (*pre-emptive multitasking*). Si es el S.O. el que gestiona los hilos, decimos que son hilos de núcleo. Por otra parte también es posible implementarlos mediante ciertos "trucos" administrándolos desde el lado de la aplicación en modo usuario, a los que llamaríamos hilos de usuario. Este tipo de hilos adolecen de ciertos problemas de importancia por lo que no se utilizan en STR, y en lo sucesivo se considerarán tan sólo los hilos de núcleo.

En la **Figura 2-13** se muestra una representación esquemática de un sistema que ejecuta varios procesos cada uno con uno o más hilos.

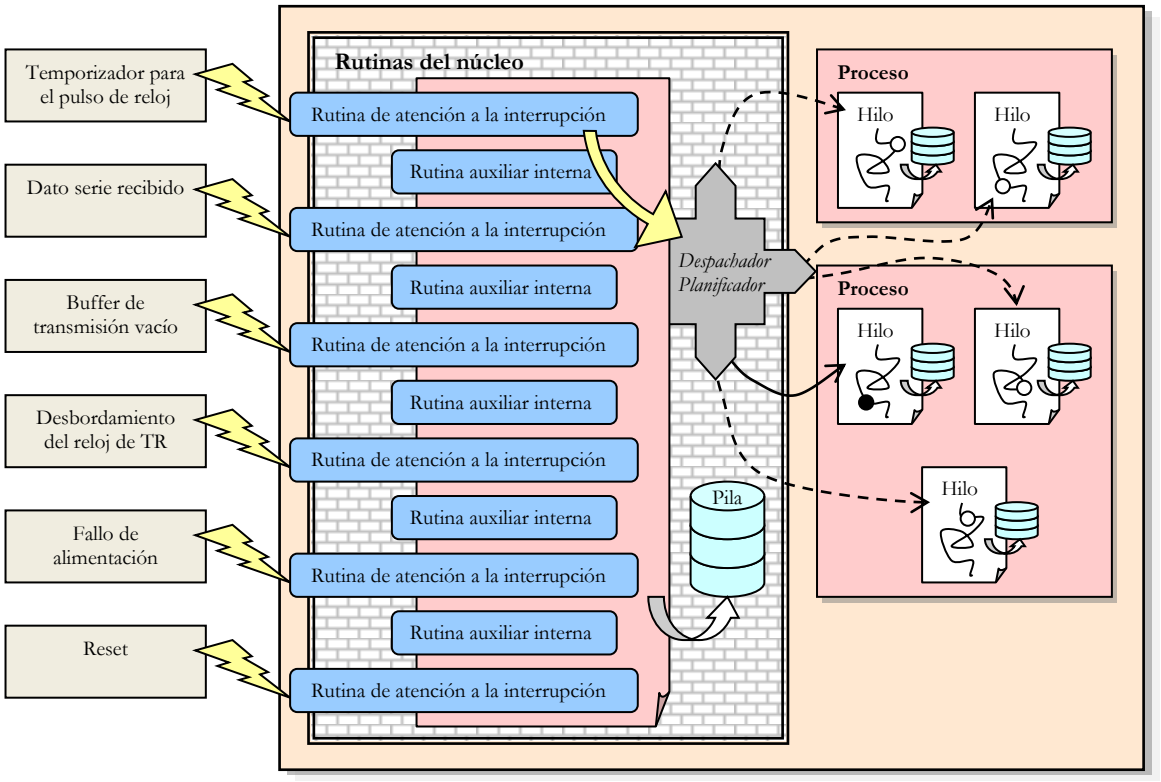


Figura 2-13. Sistema Operativo multiproceso y multihilo

En esta figura todas las rutinas del núcleo comparten una única pila accesible en modo núcleo que será diferente a cualquiera de las pilas del modo usuario usadas por los hilos. En los sistemas sencillos que no implementen soporte de modo núcleo y modo usuario, será necesario conmutar entre pilas explícitamente al entrar y salir del código del núcleo. El dibujo representa un núcleo con cierto esquema monolítico. Si se hubieran implementado las tareas menos críticas del núcleo como procesos de usuario, cada una con sus propios hilos, y cada uno de ellos usando su propia pila, se trataría de un sistema basado en micronúcleo.

Tan solo se han representado dos procesos. Los círculos huecos que aparecen en sus hilos significan el punto del código en donde se ha detenido sus ejecuciones. El círculo relleno indica el punto de ejecución del hilo que se está usando la CPU en estos momentos. Dentro del núcleo aparece el *planificador* que es el programa encargado de elegir el siguiente hilo a ser ejecutado.

2.4.4.1 Estados de un hilo

El hecho de que un hilo pueda estar a la espera de recibir su rodaja de tiempo de procesador, o bloqueado a la espera de un recurso, o en ejecución, implica que debe llevar una variable de estado asociada que refleje alguna de estas posibles situaciones. Los hilos de un núcleo real suelen tener más estados, para reflejar otras situaciones en las que se podrían encontrar.

La **Figura 2-14** muestra los estados básicos (clásicos) de un hilo:

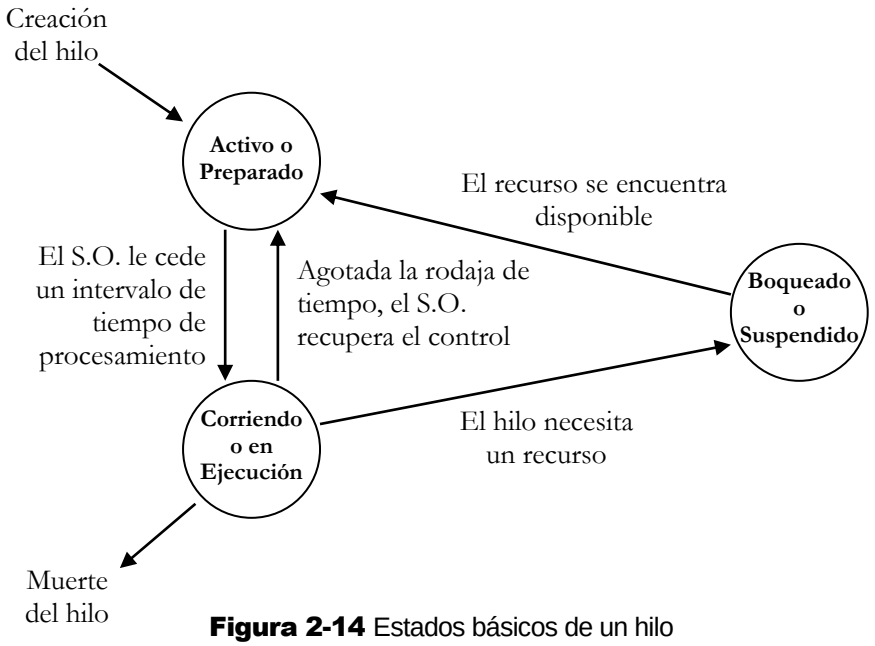


Figura 2-14 Estados básicos de un hilo

Una vez creado el hilo, pasa al estado de activo. Esto significa que el núcleo lo tiene referenciado en una cola en espera de ser ejecutado. Cuando el núcleo decide que el hilo debe correr en el procesador, lo conmuta al estado de corriendo. Se mantendrá en este estado hasta que se decida asignar el tiempo de ejecución a otro hilo, que por lo general ocurrirá cuando haya consumido el tiempo de procesamiento que el núcleo le asignó. Esto ocurrirá cuando haya consumido un tiempo de procesamiento que el núcleo le hubiera concedido, o cuando aparezca otro hilo de mayor prioridad que lo interfiera.

Durante la mayor parte del tiempo de su vida, el hilo estará conmutando entre estos dos estados, sin embargo habrá ocasiones en las que desee acceder a un recurso que no esté disponible, posiblemente por estar siendo utilizado por otro hilo. Entonces el núcleo lo configurará al estado de bloqueado, y permanecerá así hasta que el recurso vuelva a estar disponible otra vez, momento en el que pasará de nuevo al estado de activo. Cuando vuelva al estado de corriendo podrá usar el recurso.

Un hilo puede estar bloqueado a la espera de más de un recurso al mismo tiempo. Dependiendo de las intenciones del programador podría reactivarse cuando esté disponible alguno o todos los recursos por los que se espera. Es habitual por ejemplo, esperar por un recurso de hardware, y por un temporizador al mismo tiempo. El que primero llegue, reactivará al hilo.

Tener presente que un hilo bloqueado no es lo mismo que un hilo interferido aunque la consecuencia de ambos es que no se les asigna CPU: El hilo bloqueado está en una lista de bloqueados de la que el planificador no lo tendrá en consideración, mientras que el hilo interferido se encuentra en la lista de preparados, pero debido posiblemente a su baja prioridad el planificador siempre encontrará a otro mejor candidato para cederle la CPU.

La muerte del hilo normalmente será voluntaria (un suicidio), ya que lo más normal es que el propio hilo decida cuándo debe terminar. A menudo al estado de activo también se le suele llamar preparado (*ready*).

Como no todos los hilos tienen la misma importancia, los SS.OO. permiten asignarles prioridades para poder determinar cuál ejecutar en el caso en que dos o más entren en conflicto por el uso de la CPU. La *prioridad* es por tanto un valor numérico que se puede asociar a cada

hilo que gestione el núcleo cuya información indica la importancia que tendrá en la ejecución sobre los demás.

Si un hilo tiene atributos de tiempo real, tendrá prioridad sobre otros hilos que no los tengan. Dentro de estos, tendrá más preferencia aquél hilo con un valor de prioridad superior. Un hilo de baja prioridad no llegará a ejecutarse mientras existan otros hilos de tiempo real de mayor prioridad listos para ser ejecutados.

Para los hilos que no son de tiempo real, el valor de prioridad suele ser usado para asignarles un tiempo de procesamiento proporcional al mismo. Ese valor es una mera estimación que utilizará el núcleo para calcular internamente la prioridad real.

El núcleo mantendrá al menos una lista con referencias a cada uno de los hilos activos, y al menos otra con referencias a los que estén bloqueados. La estructura de datos más habitual para este almacenamiento es la cola, que consiste en una lista que permite el acceso FIFO a sus elementos. Normalmente se tendrá una cola de activos por cada prioridad admitida en el sistema, y una cola de bloqueados por cada recurso que deba ser protegido. Todo esto se verá más dilatadamente en temas posteriores.

2.4.4.2 Conmutación entre hilos

El cambio de contexto entre diferentes procesos es considerado un procedimiento costoso ya que el núcleo ha de actualizar varias estructuras de datos internas para reflejar la nueva disponibilidad de recursos para el proceso entrante, especialmente en lo referente a su particular mapa de memoria.

Como hemos visto, en un sistema basado en hilos, no se conmuta tan solo entre procesos sino que también se hace entre hilos. Un cambio de tarea que implique la conmutación entre hilos de un mismo proceso se llama *conmutación entre hilos* y tendrá una sobrecarga ligeramente inferior a conmutación entre procesos, debido a que no hay que conmutar el contexto entero, sobre todo la parte referida a la asignación de memoria del proceso. Esta es una de las ventajas de la implementación de tareas como diferentes hilos dentro de cada proceso en lugar de usar un proceso por cada tarea. Los SS.OO. que implementan hilos a nivel de núcleo, realizan una planificación entre hilos, no entre procesos.

En la **Figura 2-15** se muestran los tiempos de cambio de contexto y conmutación entre: hilos que se encuentran en un mismo proceso, e hilos en procesos diferentes. El tiempo total de cambio de contexto en los ordenadores actuales no suele exceder del microsegundo, siendo menor aún el tiempo de conmutación entre hilos.

En el dibujo aparecen tres líneas temporales indicando la ejecución de rutinas del sistema, y de los hilos de diferente prioridad que se están procesando. Téngase en cuenta que incluso los hilos pertenecientes a un mismo proceso pueden tener distintas prioridades. Los hilos τ_1 y τ_2 pertenecen al proceso 1, mientras que los hilos τ_3 y τ_4 pertenecen al proceso 2. El hilo τ_4 tiene más prioridad que τ_1, τ_2, y τ_3 por lo que el sistema favorecerá su ejecución en detrimento de la de los demás. Los tiempos de ejecución de las rutinas ISR se han representado deliberadamente grandes; en la práctica, estos son muy inferiores a los tiempos de ejecución de los hilos, que en aplicaciones reales suele ser del orden de decenas de milisegundo, al lado del ridículo tiempo empleado en la conmutación.

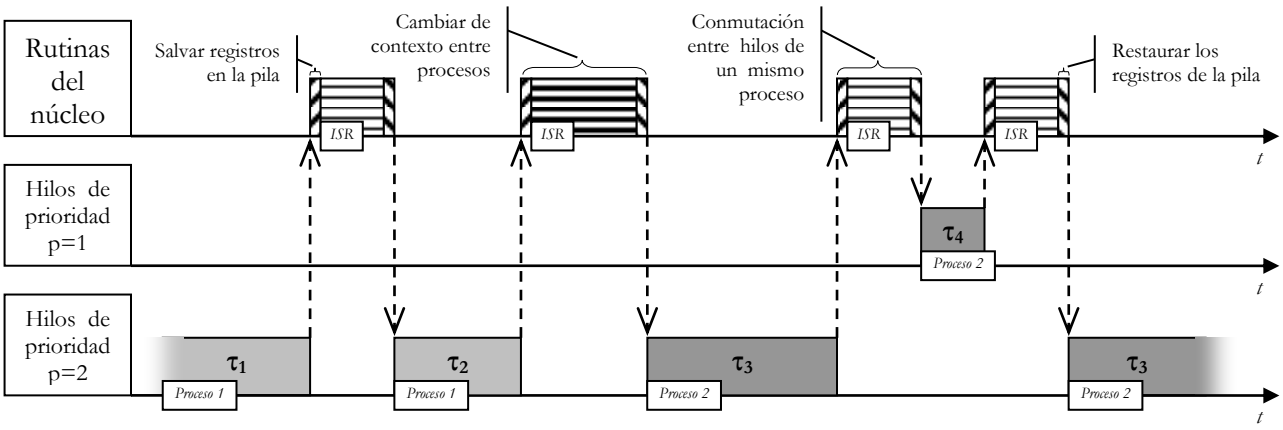


Figura 2-15 Tiempos de cambio de contexto y conmutación entre hilos

Durante el tiempo de ejecución de las rutinas del núcleo encargadas de conmutar tareas no se están ejecutando los hilos, por lo que se considera tiempo perdido, o no útil desde el punto de vista de los hilos. Esto demuestra que cuantas más conmutaciones de hilos ocurran por unidad de tiempo, el aprovechamiento del tiempo de ejecución del procesador para los hilos de usuario decae, aunque la sensación de paralelismo aumenta. En un sistema pobremente diseñado, la congestión del núcleo puede llegar hasta el punto en que se soliciten cambios de contexto con un periodo inferior al tiempo de ejecución de la ISR encargada de tal labor, con lo que se llagaría al colapso del sistema.

Las funciones **POSIX** para gestión y sincronización de hilos (*threads*) forman por separado una biblioteca de **C**. Este subconjunto de **POSIX** se llama ***POSIX**-threads* o simplemente **Pthreads**. Su objetivo es añadir algunas extensiones destinadas a la programación de Sistemas de Tiempo Real. En el diseño de las extensiones **POSIX** para TR se tuvieron dos objetivos claros que iremos desarrollando en sucesivos temas:

- Conseguir un comportamiento predecible, lo que implica ocuparse de:
  - Planificación específica de hilos.
  - Gestión de memoria virtual.
  - Señales para tiempo-real.
  - Relojes y Temporización.
- Facilidades para la programación concurrente, entre las que destacaremos:
  - Sincronización entre hilos.
  - Memoria compartida.
  - Colas de mensajes.
  - E/S síncrona y asíncrona.

2.4.4.3 Creación y destrucción de hilos

Cualquier programa que ejecutemos está compuesto al menos de un hilo de ejecución. Cuando el sistema está creando un nuevo proceso, crea un hilo inicial, desde el que comienza a ejecutarse la función `main()`.El resto de los hilos del programa han de ser creados explícitamente mediante la siguiente llamada[17]:

```
#include <pthread.h>

int pthread_create (pthread_t *tid, const pthread_attr_t *attr,
                    void *(*start_routine)(void *), void *arg);
```

- **tid**: dirección de una variable de tipo `pthread_t` que será rellenada con un identificador que referencia al hilo que se crea
- **attr**: dirección de una variable de tipo `pthread_attr_t` que contiene los atributos del hilo que se va a crear. Con `NULL` se toman los atributos por defecto
- **start_routine**: dirección de una función de tipo `void* (*)(void *)` en donde comenzará la ejecución del hilo.
- **arg**: puntero a void que será pasado como argumento de la función `start_routine`.

Al igual que ocurría con los procesos, cada hilo tiene su propio identificador que es de tipo `pthread_t`. En el momento de la creación se pueden especificar ciertos atributos de definan más concretamente las características del nuevo hilo.

Para cada nuevo hilo se debe proporcionar en `start_routine` la dirección de una función a partir de la cual comenzará su ejecución. La función `start_routine` toma como único argumento un puntero a `void` y devuelve igualmente un puntero a `void`. El último argumento de `pthread_create()` es el parámetro que se pasará a `start_routine` en el momento de ser llamada.

En el **Programa 2-9** vemos un ejemplo del uso de `pthread_create()`, en el que se ejecuta la función `func()` asincrónicamente. El hilo principal después de haber creado al hijo llamará a la función `sleep()`, pidiendo ser bloqueado durante dos segundos, pasados los cuales recuperará el estado activo; se dice que el hilo se ha echado a dormir. Lo que se pretende[18] es que el hijo haya terminado antes de que el padre se despierte.

```
/*
 * Descripción:
 * Crea un hilo y comprueba que corre.
 */

#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>

static int washere = 0;

void * func(void * arg)
{
  printf("Estoy en otro hilo diferente al principal.\n");

  washere = 1; // Las variables globales
               //  son visibles por todos los hilos.
  return 0;
}
```

[17] La especificación actual indica que los punteros que se pasen como argumentos de función que no vayan a apuntar a las mismas posiciones de memoria durante el transcurso de la misma, sean calificados con la palabra clave `restrict`. Esto se hace para informar al compilador de antemano y darle la oportunidad de optimizar el código que genere. Por claridad se omitirá esta palabra clave en lo sucesivo.

[18] POSIX no asegura que esto vaya a ser así en cualquier circunstancia. Más adelante veremos cómo asegurarnos de que un hilo haya terminado.

```
int main()
{
  pthread t t;

  printf("Estoy en el hilo principal.\n");

  pthread create(&t, NULL, func, NULL);

  sleep(2); // El hilo principal duerme durante 2s

  if(washere == 1)
       printf("Otro hilo diferente al principal ha modificado "
              "la variable global.\n");

  return 0;
}
```

Programa 2-9. Creación de un hilo.

Si todo fuera bien, la salida de este programa sería:

```
Estoy en el hilo principal.
Estoy en otro hilo diferente al principal.
Otro hilo diferente al principal ha modificado la variable global.
```

A continuación se muestra una representación grafica que muestra el flujo de ejecución del programa:

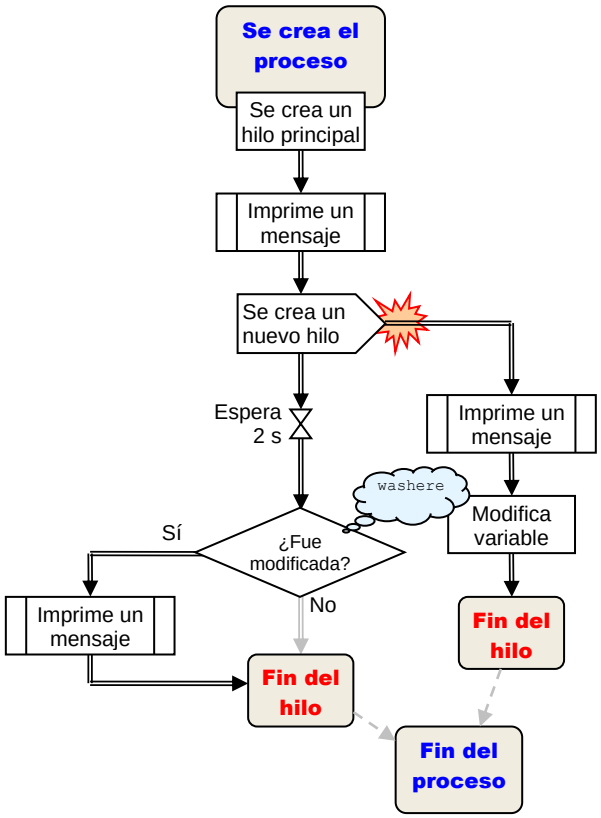


Figura 2-16. Evolución concurrente de dos hilos de un mismo proceso.

Desde la función `func()`, podríamos hacer llamadas a otras funciones, que se ejecutarán en paralelo con el hilo principal.

Cada uno de los hilos terminará cuando regrese la función en donde comienza su ejecución. Si quisiéramos, podríamos salir prematuramente de un hilo antes del retorno natural de la función, haciendo una llamada en cualquier momento a `pthread_exit()`:

```
#include <pthread.h>
int pthread_exit (void *value_ptr);
```

El argumento `value_ptr` es devuelto en lugar del que acompaña al `return` de la función correspondiente:

```
/*
 * Descripción:
 * Realiza una llamada explítica para terminar el hilo principal.
 */

#include <pthread.h>

int main()
{
        // Salir de este hilo
        pthread exit((void *) 3); // Equivalente a: return 3;
                                  //                exit(3);

        // Lo siguiente no se llegará a ejecutar nunca
        return 2;
}
```

Programa 2-10. Terminación explícita del hilo principal.

Observar que el flujo que ejecuta `main()` se comporta como un hilo más. Igualmente podríamos haber salido de cualquier otro hilo:

```
/*
 * Descripción:
 * Realiza una llamada explítica para terminar el hilo hijo
 */

#include <pthread.h>
#include <stdio.h>
#include <time.h>

void *func(void * arg)
{
  printf("Hola.\n");

  pthread exit(arg); // Usamos el mismo argumento para el código devuelto

  printf("Adiós.\n"); // Esto no se ejecutará

  return (void *) 0x323A; // Esto no se ejecutará
}

int main()
{
  pthread t t;

  pthread create(&t, NULL, func, NULL);

  sleep(1);

  return 0; // Equivalente a pthread exit(0), y a exit(0)
}
```

Programa 2-11. Terminación explícita de los hilos.

Al crear un nuevo hilo, se le devuelve un identificador al que lo creó, para que lo pueda usar en posteriores llamadas. Para que un hilo pueda obtener su propio identificador usamos la función `pthread_self()`:

```
#include <pthread.h>
pthread_t pthread_self (void);
```

En el caso en que quisiéramos saber si dos identificadores distintos referencian al mismo hilo, no podemos usar el operador de comparación == de **C**, ya que se tratan de tipos de datos abstractos. En su lugar debe ser usada la función `pthread_equal()` que devolverá un valor verdadero en el caso en que sus dos argumento referencien al mismo hilo, y FALSE en caso contrario.

```
#include <pthread.h>
int pthread_equal (pthread_t t1, pthread_t t2);
```

En el siguiente código se muestra el uso de `pthread_equal()`.

```
#include <pthread.h>
#include <assert.h>
#include <stdio.h>

pthread_t t2;

void * func(void * arg)
{
  pthread_t t;

  sleep(2);

  t = pthread_self();

  if(pthread_equal(t, t2) == 0)
        puts("Los identificadores sí son iguales.");
  else
        puts("Los identificadores no son iguales.");

  return 0;
}

int main()
{
  pthread_t t1;

  assert(pthread_create(&t1, NULL, func, (void *) 1) == 0);

  assert(pthread_create(&t2, NULL, func, (void *) 2) == 0);

  assert(pthread_equal(t1, t2) == 0);

  assert(pthread_equal(t1, t1) != 0);

  sleep(4);

  return 0;
}
```

La macro `assert()` interrumpe la ejecución del código si el argumento que se le pasa es falso.

2.4.4.4 Ejecución asíncrona

En un sistema bien diseñado, cada hilo avanza por su propia secuencia de instrucciones lo que en cierto sentido marca una evolución de su particular percepción del paso del tiempo. Esta será independiente de la existencia concurrente de otros hilos. El acceso a código y recursos comunes entre varios hilos ha de ser transparente desde la perspectiva del hilo. En

tiempo de desarrollo[19], el programador sabe el orden en que se ejecutará la secuencia de instrucciones que ha escrito, al menos a grandes rasgos[20].

Si se conoce a priori el orden de ejecución de la secuencia de instrucciones del código que procesa un hilo, decimos que se están procesando de forma *síncrona*. Esto es independiente de la existencia de otras tareas concurrentes, o de la determinación del momento exacto en que se vayan a ejecutar cada una. Un claro ejemplo de tales secuencias son las llamadas a funciones de cálculo matemático que aparecen en el **Programa 2-12**:

```
/*
 *  Programa que opera haciendo uso únicamente de funciones síncronas.
 */

#include <math.h>

double AbsDouble(double num) // Calcula el valor absoluto de un número
{
    return num < 0.0 ? -num : num;
}

double Calcula(double num) // Realiza varios cálculos matemáticos
{
    double res;

    res -= 1.0;
    res += 2 * sin(num); // Añade el doble del seno de num
    res = AbsDouble(res);
    res = lgamma(res); // Calcula la función gamma del argumento

    return res * 24.0;
}

// El programa recibe como único parámetro un número en coma flotante
int main(int argc, char* argv[])
{
    double dArg, dRes;

    if(argc < 2) // Comprobar que se ha introducido suficientes parámetros
        return -1;

    dArg = atof(argv[1]); // Convierte la cadena de texto a un número double

    dRes = Calcula(dArg); // Realiza procesamiento

    // Operar con el resultado dRes devuelto
    // . . .

    return 0;
}
```

Programa 2-12. Uso de funciones síncronas.

Si se analiza el programa se sabrá exactamente el orden de ejecución de la secuencia de instrucciones. Las subrutinas llamadas sólo regresarán cuando hayan sido completamente procesadas. Las funciones `AbsDouble()` y `Calcula()` son por tanto de ejecución síncrona, al igual que las instrucciones que contienen.

Existe por otra parte un tipo de subrutinas que inician una operación, y regresan inmediatamente antes de haberla concluido. Esto significa que internamente debe existir algún tipo de concurrencia, que permita la realización de la operación en paralelo. La aplicación que efectúa la llamada podría seguir realizando sus quehaceres, para posiblemente realizar más

[19] Considerando el uso de un lenguaje de programación imperativo como C o Ensamblador.

[20] Los compiladores podrían alterar ligeramente el orden original de ejecución de instrucciones máquina en la fase de optimización de código.

adelante otra llamada con el objeto de confirmar que ha concluido la operación que se inició. Decimos que esa subrutina se ejecuta *asincrónicamente*.

En el **Programa 2-13** se ejecuta una misma función de forma síncrona y asíncrona. En el caso asíncrono se ha utilizado un hilo para permitir su ejecución concurrente.

```
/*
 *  Comparación de ejecución síncrona y asincrónica.
 */
#include <math.h>
#include <time.h>
#include <pthread.h>

double g dNum;

void *CalculoTrig(void * arg)
{
  int i;

  for(i = 0; i < 1000; i++)
  {
    g_dNum = sin(g_dNum);
    g dNum = cos(g dNum);
    g dNum = tan(g dNum);
    g dNum = atan(g dNum);
    g_dNum = acos(g_dNum);
    g_dNum = asin(g_dNum);
  }

  return 0;
}

int main()
{
  pthread t t;

  g dNum = 0.567;

   // Inicia una ejecución síncrona de CalculoTrig
  CalculoTrig(NULL);

  if(g dNum == CONSTATE INICIAL)
      printf("El resultado %g no ha degenerado.\n", g dNum);
  else
      printf( "El resultado %g ha degenerado respecto a %g.\n",
              g dNum, CONSTANTE INICAL);

  // Inicia una ejecución asíncrona de CalculoTrig
  pthread_create(&t, NULL, CalculoTrig, NULL);

  // No podemos saber cuándo terminará de ejecutarse CalculoTrig en este caso !!!

  sleep(1); // Duerme durante 1 segundo

  return 0;
}
```

Programa 2-13. Conversión de una función síncrona en asíncrona.

En la mayoría de las ocasiones las llamadas asíncronas están dirigidas a dispositivos de E/S con respuestas relativamente lentas. De esta forma se permite que la aplicación siga procesándose aunque no se haya consumado el acceso a dichos dispositivos. Existen multitud de funciones de ejecución asíncrona como `printf()`, `puts()`, `fwrite()`,... que realizan internamente llamadas al sistema para acceder al hardware en diferido. En tales casos se suele incluir funciones como `fflush()` que esperan a que termine el acceso al hardware consiguiendo así el sincronismo.

Ejercicios

15. Definir brevemente los siguientes conceptos relacionados con la informática:

Código máquina.

Es el código de instrucciones que entienden directamente los procesadores.

Programa.

Es una secuencia de órdenes dispuesta para ser ejecutada por un procesador. Un programa debe especificar su propio flujo de ejecución, incorporando la posibilidad de realizar saltos, y bucles. Además debe poder tomar decisiones basadas en resultados previos.

Rutina.

Es un programa informático normalmente de pequeñas dimensiones que tiene un objetivo concreto y que puede hacer uso de subrutinas. Habitualmente se consideran rutinas a los programas que no son llamados explícitamente por otros programas de usuario, sino por el propio sistema. También a pequeñas secuencias de instrucciones contenidas en una subrutina.

Subrutina.

Una secuencia de un programa especialmente concebida para poder ser reutilizada mediante llamadas explícitas desde diferentes partes del mismo. Al finalizar la ejecución de la subrutina, se debe continuar con la de la rutina que provocó el salto.

Función de C.

Es la forma que dispone el lenguaje C para realizar subrutinas. Una función de C además implica una normalización a bajo nivel que permite usar un número variable de argumentos de entrada, devolver valores, y almacenar variables locales entre otras cosas. De esta forma se puede acceder a las funciones mediante una interfaz de alto nivel que oculta los detalles de implementación.

Rutina de fondo.

Se trata de un programa de baja prioridad que es ejecutado cuando no se tienen otras tareas más importantes que procesar.

Rutina de atención a interrupción (ISR).

Es la rutina que se llama automáticamente en respuesta a una interrupción.

Tarea.

Es un programa en ejecución que tiene una finalidad concreta. Dispone de sus propias zonas de almacenamiento de datos independientes de las de otras tareas, con las que por otra parte puede intercambiar información.

Hilo.

Es una tarea que dispone de su propia pila de procesador. Comparte espacio de direccionamiento y recursos con otros hilos pertenecientes al mismo proceso. El sistema puede detener su ejecución y reanudarla de acuerdo a políticas de planificación internas. También son llamados *procesos ligeros* (LWP).

Hilo a nivel de usuario.

Hilo cuya decisión de conmutación del uso de CPU a otro hilo es tomada a nivel de la aplicación de usuario, la cual debe disponer de algún mecanismo de interrupción temporizada. El S.O. no es consciente de su existencia, por lo que el hilo no puede beneficiarse de la planificación que el núcleo pudiera hacer. La conmutación entre un hilo y otro es ágil, al no requerir la intervención del núcleo.

Hilo a nivel de núcleo.

Hilo cuya decisión de conmutación del uso de CPU a otro hilo es tomada a nivel de núcleo de S.O. El núcleo pude tener en cuenta la posible prioridad y bloqueo de recursos que mantenga, lo que redunda en una mejor planificación. También asignará más adecuadamente los tiempos a los hilos que forman un mismo proceso.

Proceso.

Ente creado por el S.O. para contener los recursos necesarios por una aplicación. Los elementos característicos más importantes que lo componen son: hilos, memoria para almacenar tanto el programa como los datos internos que manejan los hilos (especialmente la pila de cada uno de ellos), referencias a todos los objetos del sistema que se estén usando por sus hilos (descriptores de archivos y dispositivos, asignaciones de memoria compartida,…). Tradicionalmente los procesos sólo contaban con un hilo o tarea de ejecución, por lo que la literatura antigua no hacía distinción entre hilo, tarea y proceso. Para diferenciarlos de los procesos ligeros se les llama *procesos pesados*.

Aplicación de usuario.

Programa destinado a ofrecer servicios de alto nivel a los usuarios o a otras aplicaciones de usuario. Cuentan con el nivel de privilegio menor que se pueda disponer a nivel de CPU, y se ejecutan en un entorno muy protegido.

Software del sistema.

Programas de bajo nivel destinados a simplificar el acceso al hardware y administrar los datos. Funcionan con el mayor nivel de privilegio de CPU, lo que les permite realizar funciones críticas. En el caso de existir S.O. lo más representativo es su núcleo.

Como significado adicional, el software del sistema también comprende a las herramientas de desarrollo de bajo nivel como compiladores de **C**, analizadores de gramáticas, etc.

2.4.5 Gestión avanzada de memoria

Como sabemos la memoria almacena la información que necesitan las tareas, formada tanto por el código de los programas, como por los datos que manejan. Con el objetivo de

conseguir mayor fiabilidad, el sistema ha de disponer de algún mecanismo de seguridad para evitar que tareas con comportamientos anómalos o malintencionados, puedan corromper la memoria correspondiente a las demás.

Una posibilidad básica de protección la encontramos en dispositivos pequeños en los que la cantidad de código que se utiliza es relativamente baja y no se va a modificar durante el funcionamiento normal, como en el caso de los microcontroladores. Consiste en grabar los programas en memorias de sólo lectura, y usar una memoria RAM para almacenar el estado interno del sistema. Con esto nos aseguramos que no se podrán modificar involuntariamente los programas durante el funcionamiento normal del sistema. No obstante este sistema no protege de la modificación indeseada de las áreas de RAM.

Para preservar la seguridad de accesos no autorizados a las diferentes zonas del mapa de memoria, y facilitar la administración de memoria por parte del sistema, se han diseñado varias técnicas ingeniosas. En la actualidad la mayor parte de los sistemas de tamaño medio y grande utilizan una técnica llamada *paginación de memoria*.

La paginación de memoria requiere el uso de una circuitería especial llamada Unidad de Administración de Memoria (*Memory Management Unity - MMU*) que funciona en estrecha colaboración con la CPU, hasta tal punto que en los diseños actuales forma parte inseparable del mismo chip. La principal misión de la MMU es crear un espacio de *direcciones lógicas* de memoria, diferente al espacio de *direcciones físicas* que se utilizaba tradicionalmente. Para esto, el S.O. rellena unas tablas de traducción de direcciones que emplea la MMU, para transformar dinámicamente cualquier dirección presente en el bus interno de la CPU (dirección lógica), por otra correspondiente a un área de memoria distinta (dirección física).

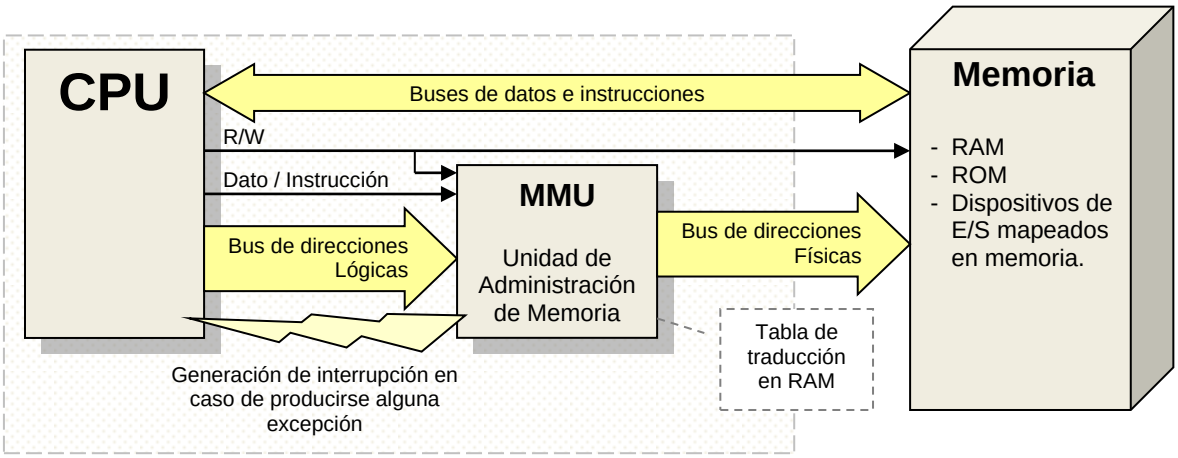


Figura 2-17. Disposición de la Unidad de Administración de Memoria

El objetivo es que cada proceso pueda disponer de un mapa de direcciones de memoria lineal e independiente a los del resto. Con este sistema varios procesos podrían creer en apariencia que acceden a la misma posición de memoria cuando en realidad acceden a direcciones físicas distintas, hasta este punto llega el aislamiento entre ellas. Es misión del S.O. actualizar las tablas de traducción cada vez que se produzca un cambio de contexto, lo que le añade cierta penalización temporal. Como el almacenar en la tabla de traducción una dirección física por cada dirección virtual sería inviable, los dos espacios de direcciones lógicas y físicas se dividen en bloques llamados *páginas de memoria*. El tamaño de cada página es habitualmente de 4 u 8 kilobytes. A cada página lógica se le hace corresponder con una física, mediante un offset que la MMU añade a la dirección que genera la CPU. Así el tamaño de la tabla es razonablemente pequeño y manejable. Las tablas se almacenan en memoria RAM, aunque se benefician de la caché para acelerar su lectura. Cualquier intento de acceder a una dirección que

no corresponda con alguna entrada de página lógica de la tabla de traducción vinculada al proceso que solicitó el acceso, generará una excepción, lo que le da la oportunidad al S.O. de tomar las medidas oportunas.

En el caso de los x86, de los 32 bits efectivos con los que cuenta el bus de direcciones, los 20 bits de mayor peso son los únicos utilizados por la MMU para buscar una entrada en la tabla de traducciones. El tamaño máximo de la misma será por tanto de 2^{20} (más de un millón) de entradas, de las que sólo serán utilizadas el mínimo número imprescindible para representar el mapa de direcciones virtuales del proceso. Los 12 bits de menor peso pasan directamente al bus de direcciones físicas. Se puede considerar que estos forman un offset de 4096 bytes dentro de cada página.

Es de importancia destacar el hecho de que no es un requerimiento exigible ni posiblemente deseable que los STR se basen en sistemas informáticos que hagan uso de MMUs. El tener que trabajar con ellas es una consecuencia de pretender usar elementos de la informática de consumo para construir STR sobre ellos. Es importante saber cómo funciona el sistema de paginación de memoria para poder aprovecharlo o anularlo según convenga.

2.4.5.1 Memoria lógica y física

El uso de memoria paginada le aporta al sistema un robusto mecanismo de seguridad en los accesos no autorizados. Por cada página referenciada en las tablas se reserva un campo de información adicional útil para el S.O. en donde se especifican los permisos que tiene el proceso sobre la página: lectura, escritura, y ejecución.

En la parte inferior de la **Figura 2-18** se muestra la memoria RAM de un computador dividida en 16 páginas iguales. En ella se pretende almacenar las rutinas del núcleo del S.O., el código y los datos pertenecientes a los procesos, y espacio para la caché de disco duro. Como vemos, el área reservada para aplicaciones aparece conteniendo páginas desordenadas correspondientes a los procesos. En la parte superior figuran los espacios de memoria de las tareas de usuario correctamente ordenados.

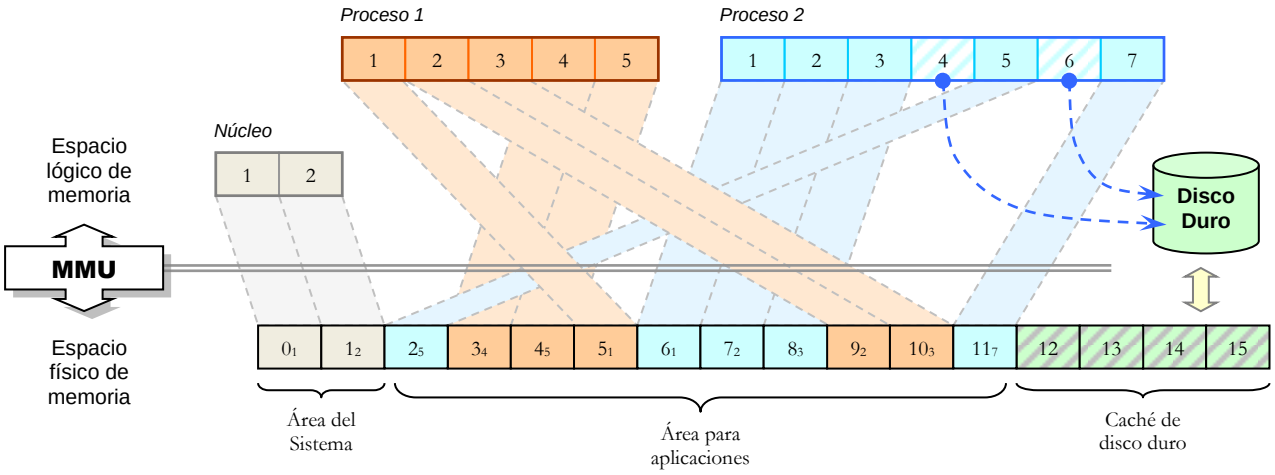


Figura 2-18. Disposición figurada de la correspondencia de páginas lógicas con páginas físicas

La MMU se encarga de realizar la asignación entre la memoria lógica y la física lo que permite que las aplicaciones perciban su mapa de memoria como si dispusieran de un espacio lineal e independiente al de las demás.

Tal y como aparece en la **Figura 2-18**, la memoria lógica del núcleo podría corresponderse directamente con la memoria física. En este sentido la MMU no se utiliza para realizar traducciones de la memoria del núcleo.

En el caso de las aplicaciones representadas por los procesos 1 y 2, el desorden de las páginas físicas se debe a que la memoria fue concedida según ha ido siendo requerida, para lo que se asignaron las páginas que en el momento de la solicitud estuvieran disponibles. En cualquier sistema real, pasado un rato desde que las aplicaciones comenzaran a ejecutarse, y debido a las frecuentes peticiones y liberaciones de memoria, las páginas físicas terminarán por estar completamente desordenadas. Las tareas de tiempo real no influyen demasiado en la fragmentación de memoria, ya que como más tarde veremos, deben realizar las peticiones de recursos al principio de su ejecución, bloquearlos una vez concedidos, y liberarlos al final.

2.4.5.2 Memoria virtual

Los SS.OO. modernos permiten que la cantidad total de memoria utilizada por los programas sea superior a la que se encuentra físicamente instalada, lo que es económicamente interesante para el usuario medio. Para conseguir esta ilusión, cuando el núcleo recibe peticiones de nuevas asignaciones de memoria, ordena llevar al disco duro las páginas que no se hayan estado utilizando en un pasado reciente. Si se necesitaran en un futuro, serían llevadas de nuevo a RAM desde el disco duro. Al proceso de copiar páginas del el disco duro a la RAM y viceversa se le llama *intercambio de páginas*. Normalmente el S.O. reserva un espacio en el disco duro por cada página de memoria que pueda asignar para las aplicaciones de usuario. La mayoría de estas reservas son realizadas sobre un archivo especial llamado *archivo de intercambio* (*swap*). No existe garantía alguna para asegurar que una página que fue llevada a swap[21] sea más adelante vuelta a mapear[22] en la misma página física. En la figura, las páginas lógicas 4 y 6 del proceso 2 no existen en la memoria física, por lo que cabría de esperar que estuvieran almacenadas temporalmente en el archivo de intercambio del disco duro. La memoria adicional que se consigue con este sistema se llama *memoria virtual*.

Por último aparece en la figura una zona reservada para *cachear* al disco duro[23], área que suele llevarse un espacio importante de la memoria física de los sistemas. Este espacio se utiliza para acelerar los accesos al disco duro: a menudo en lugar de hacer operaciones de E/S directamente con el disco, se realizan con esta memoria como si se tratara del propio disco duro. Cuando el S.O. lo estime necesario actualiza los contenidos del disco duro con los de esta memoria, para lo que necesita tan solo unas pocas operaciones de E/S. La ganancia de velocidad en las operaciones se consigue porque se evita tener que realizar multitud de lentas pequeños accesos directos al disco, tal y como son solicitadas por las aplicaciones. El intercambio de páginas correspondientes a la memoria virtual también se acelera mediante el uso de esta caché.

Al mejorar las operaciones de intercambio de páginas, la caché de disco duro consigue disminuir notablemente el tiempo medio de respuesta de las aplicaciones que no encajen completamente en RAM. No obstante, el precio que imponen estas técnicas dispara los tiempos máximos de respuesta, por lo que su uso está vetado en tareas críticas de tiempo real.

[21] "Llevar a swap" es una expresión utilizada en ocasiones para indicar que se ha llevado información al archivo de intercambio. También se utiliza su contraria "traer de swap" con significado inverso.

[22] Mapear es una expresión muy usada que significa hacer que aparezca secciones de memoria física en unas determinadas posiciones del mapa de memoria lógica de un proceso.

[23] Es importante diferenciar claramente la caché del procesador y la caché del disco duro. La primera es hardware, y la segunda se realiza mediante software.

2.4.5.3 Memoria de un proceso

Como ya sabemos, uno de los recursos más característicos de todo proceso es su asignación de memoria, que es compartida por todos los hilos que contiene. Las partes fundamentales del mapa de memoria de un proceso son:

- Área[24] de código, desde donde se lee el código máquina de los programas que se están ejecutando. También es conocida como área de texto.
- Área estática de datos, donde se almacenan todos los datos que hayan sido reservados en tiempo de compilación/ensamblado. Se compone a su vez de:
  - Datos previamente iniciados de sólo lectura.
  - Datos previamente iniciados de lectura-escritura, pe. las variables globales de los programas de lenguaje **C**.
  - Datos no iniciados. Suelen ser reservas de memoria para uso interno, habitualmente rellenadas por el programa cargador.
- Área dinámica, que almacena cualquier tipo de información que sea reservada durante la ejecución de los hilos. Sus principales usos son el almacenamiento de:
  - El montón de memoria (*heap*). Se trata de la memoria que se va reservando mediante solicitudes explícitas de memoria a `malloc()` y derivadas. El S.O. la va concediendo con una granularidad de páginas enteras, siendo función de las bibliotecas de usuario el administrarla a mayor resolución.
  - Pilas para los hilos. Según se van utilizando, las pilas pueden crecer sobrepasando la capacidad de las páginas que se les había estimado. En tales casos, el sistema les puede conceder automáticamente nuevas páginas según se vayan solicitando. Normalmente se reservan en posiciones distantes de memoria para permitir que puedan crecer sin llegar a solaparse.
  - Espacio para memoria compartida entre procesos.
  - Espacio para mapear imágenes archivos en memoria.
  - Espacio para mapear imágenes de otros archivos binarios ejecutables, pe. bibliotecas de enlace dinámico. Esta área es tratada de un modo especial por el S.O.

El mapa de memoria [lógica] de un proceso de usuario de tres hilos sería similar al de la **Figura 2-19**.

[24] A las áreas de memoria también se les suele llamar *segmentos*. No se va a emplear ese nombre para que no exista confusión con los segmentos de memoria de los x86.

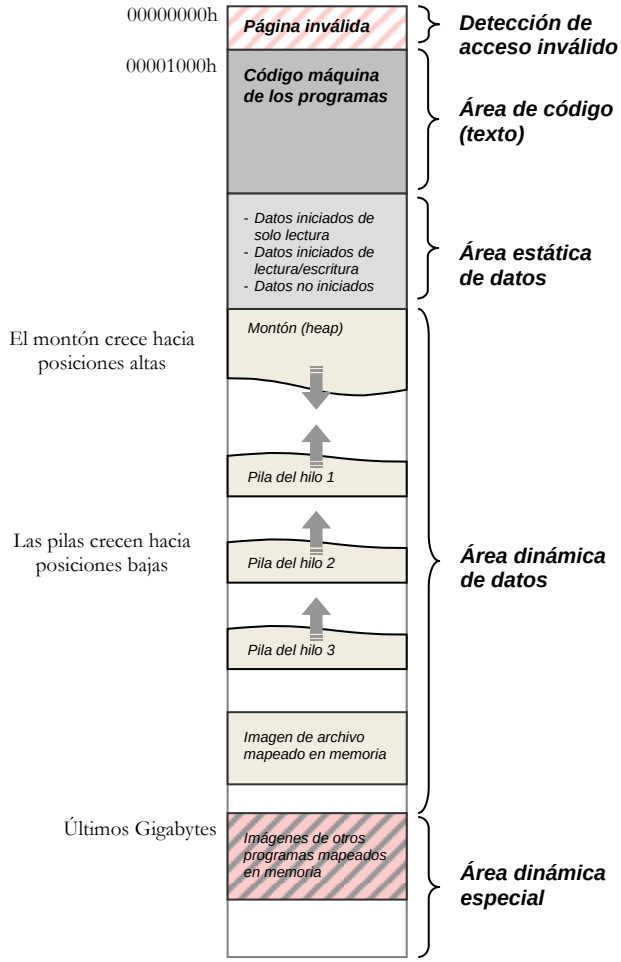


Figura 2-19. Mapa de la memoria lógica de un proceso

En la parte superior aparece una página sin contenido por la que se accedería a los 4 primeros kilobytes lógicos. Habitualmente a esta página no se le hace corresponder con memoria física alguna para que cuando se intente acceder a ella, la MMU genere una excepción por acceso inválido. Con ello se intenta capturar cualquier intento de acceso por medio de un puntero NULL, que como sabemos apunta a la posición 0. Existen SS.OO. como **Windows**, que reservan una sección aún más grande. En ese caso se hace para capturar los intentos de acceder directamente al hardware del PC como se hacía antiguamente. El área inválida es de 4 MB en ese caso.

A continuación hay un espacio para almacenar una imagen de código del programa. Este es a menudo más grande que el resto de espacios reservados para datos. En el programa se emplean direcciones absolutas para los saltos y el acceso al área estática de datos. Estas páginas son marcadas con los atributos de lectura y ejecución.

El área estática de datos almacena todos los datos que se definieron estáticamente en tiempo de compilación o ensamblado. Una rutina especial se encarga de rellenar los contenidos de los datos que contengan información inicial, como son los valores iniciales de las variables globales. Las páginas que contengan datos que no puedan ser reescritos en tiempo de ejecución, serán marcadas como de *sólo lectura*.

Inmediatamente después suele aparecer el llamado montón de memoria, que almacena asignaciones dinámicas de memoria. Puede crecer y decrecer con el tiempo dependiendo de la marcha del programa.

A continuación y con cierto espacio de separación entre ellas, aparecen las pilas de los diferentes hilos. Apréciese que crecen en sentido opuesto al del montón.

Por último pueden aparecer reservas adicionales de memoria para:

- Contener archivos del disco duro que se hayan proyectado en la memoria lógica para beneficiarnos en su acceso del sistema de memoria virtual.
- Contener la imagen de otros programas que hayan sido mapeados en memoria después de que se creara el proceso. Este espacio suele ser ocupado por las bibliotecas de funciones compartidas o de enlace dinámico. Estas no suelen contener referencias a posiciones absolutas de memoria ya que hasta el momento de ser cargadas, no se sabe dónde serán finalmente alojadas. Son útiles para incorporar funcionalidad no presente en el programa principal al estilo de *plug-ins*.
- Poder acceder a memoria compartida entre diferentes procesos.

En el **Programa 2-6** vimos que la llamada al sistema `fork()`, sirve para clonar a un proceso. Es posible pensar en la ineficiencia que tendría esta función ya que la clonación de programas grandes acabaría pronto con el espacio de memora. Sin embargo debido a la magia de la MMU, es posible que varios procesos compartan el mismo área de código. El núcleo simplemente tendría que asignar las mismas páginas físicas a las páginas lógicas correspondientes de los procesos, tal como muestra la **Figura 2-20**. El S.O. **Windows** no utiliza en sus versiones más modernas este mecanismo de compartición de código.

En este caso, es posible que el proceso 2 sea un clon del proceso 1. El proceso 1 podría haber creado un nuevo hilo después de la clonación, por lo que tiene dos pilas, mientras que el proceso 2 podría haber reservado más memoria. Las casillas que aparecen rayadas corresponden a páginas que no tienen correspondencia en la memoria física debido tal vez a que no hayan sido accedidas recientemente, por lo que se habrían llevado al archivo de intercambio del disco duro. Los procesos no sólo pueden compartir memoria para código, sino que mediante llamadas al sistema pueden también compartir memoria para datos.

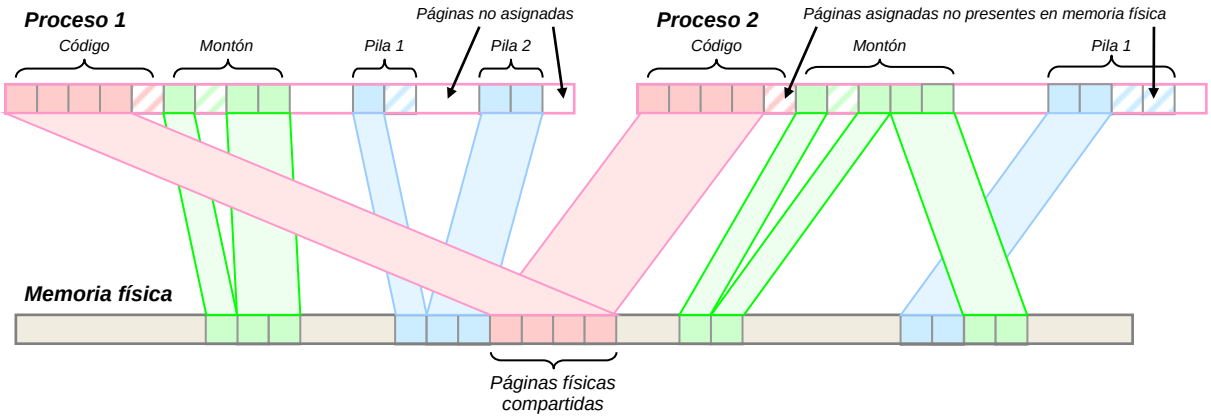


Figura 2-20. Dos procesos compartiendo las mismas páginas físicas de memoria para código

Los sistemas informáticos que almacenan el código de las aplicaciones en memoria RAM no suelen impedir que los procesos modifiquen el código máquina durante su ejecución. En el caso de que las páginas físicas sean compartidas se utiliza un algoritmo llamado

copy-on-write, que realiza una copia de la página modificada para hacerla sólo accesible al proceso que realizó la modificación de código.

Veamos ahora un ejemplo en el que un proceso crea a otro a partir de una imagen de archivo en disco distinta. Para ello se puede utilizar la familia de llamadas `exec`: `execv`, `execl`, `execve`, `execle`, `execvp`, y `execvl`. Todas ellas hacen uso del programa cargador (*loader*) del S.O., cuyo cometido es sustituir el mapa de memoria del proceso por el que se se especifica en un archivo ejecutable del disco, liberando o reasignando memoria según sea necesario. Los ejecutables que generan los compiladores están acompañados de una cabecera con toda la información necesaria para que el cargador sepa recrear el mapa de memoria del nuevo proceso. En el caso de **GNU/Linux**, se trata del la cabecera ELF (*Executable and Linking Format*), y en caso de **Windows**, de la PE (*Portable Ejecutable*).

```
#include <unistd.h>

int execl (const char *filename, const char *arg0, ...);
```

- **filename**: nombre del programa ejecutable a cargar.
- **arg0**: cadena de texto que recibirá la aplicación como primer argumento argv[0] en
  int main(int argc, char* argv[])
- **…**: otras cadenas de texto correspondientes a los argv[1], argv[2],… Poner un NULL en el último argumento.

```
/*
 *  Creación de un nuevo proceso cargando su imagen desde el disco.
 */
#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

// Poner aquí el nombre de un programa presente el el sistema de archivos:
#define NOMBRE_PROGRAMA "/opt/Acrobat/acroread"

int main ()
{
  int status;
  pid_t pid;

  pid = fork (); // Clona al proceso actual
  if (pid == 0)
  {
      // Este es el proceso hijo. Ejecuta el programa:
      execl (NOMBRE_PROGRAMA, NOMBRE_PROGRAMA, NULL);

      // Si todo ha ido bien, la siguiente línea no se debería llegar a ejecutar.
       exit (EXIT FAILURE); // Para compatibilidad ISO usar  Exit() en lugar de  exit()
  }
  else if (pid < 0)
  {
    // La clonación ha fallado. Informar del error
    status = -1;
  }

  // Continuar la ejecución del programa
  // . . .

  return status;
}
```

Programa 2-14. Carga de la imagen de un nuevo proceso.

El código `EXIT_FAILURE` que devuelve `_exit()` es una constante predefinida que indica que ha ocurrido un fallo.

Como ejemplo adicional en la documentación de GNU aparece un código que bien podría ser el de la orden `system()`:

```c
/*
 *  Creación de un nuevo proceso desde el disco, imitando el
 *   funcionamiento de la órden system()
 */
#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

// Ejecutar el comando usando esta shell
#define SHELL "/bin/sh"

int mi system (const char *command)
{
  int status;
  pid t pid;

  pid = fork (); // Clona al proceso actual
  if (pid == 0)
    {
      // Este es el proceso hijo. Ejecuta el comando de la shell
      execl (SHELL, SHELL, "-c", command, NULL);
       exit (EXIT FAILURE); // Para compatibilidad ISO usar  Exit() en lugar de  exit()
    }
  else if (pid < 0)
    // La clonación ha fallado. Informar del error
    status = -1;
  else
    // Este es el proceso padre. Esperar a que termine el hijo.
    if (waitpid (pid, &status, 0) != pid)
      status = -1;

  return status;
}
```

Programa 2-15. Funcionamiento interno de la función `system()`

Se trata de una función que crea una nueva *shell*, y le pide que intérprete el comando que se le envíe. Si el comando es un nombre de programa, será convenientemente cargado. En este caso el proceso padre esperará a que termine el hijo mediante la llamada a `waitpid()`, haciendo que la ejecución del mismo ocurra sincrónicamente.

Nota

La función `fork()` tan sólo clonará al hilo que la llame, pero el área de memoria que utilizaran otros hilos en la que se incluye a las pilas sí será duplicada, cosa que posiblemente no sea lo que se pretende. Si se quiere clonar un proceso lo mejor es hacerlo antes de haber creado nuevos hilos.

El empleo de la pareja `fork()` – `exec()` requiere el uso de un sistema hardware con soporte de MMU, y muy seguramente también un sistema de intercambio de páginas con el disco duro. Esto es un problema para el desarrollo de STR ya que no están obligados a usar ni una cosa ni otra. **POSIX** propone las funciones siguientes que no implican el uso de mecanismos avanzados para su creación:

```c
#include <spawn.h>

int posix spawn( pid t *restrict pid, const char *restrict path,
                 const posix_spawn_file_actions_t *file_actions,
                 const posix_spawnattr_t *restrict attrp,
                 char *const argv[restrict], char *const envp[restrict]);
int posix spawnp( pid t *restrict pid, const char *restrict file,
                  const posix spawn file actions t *file actions,
                  const posix spawnattr t *restrict attrp,
                  char *const argv[restrict], char * const envp[restrict]);
```

A día de hoy estas funciones no están muy extendidas.

Ejercicios

16. Suponiendo que el procesador *picoJava* tiene una longitud media de instrucción máquina de 1,8 bytes, y que el procesador *Itanium* la tiene de 5,3 bytes. ¿En qué mediada afectará este factor si ambos tienen idénticas capacidades de caché?

Este factor afectará negativamente a la probabilidad de encontrar en caché instrucciones que se hayan usado hace poco, más en concreto en la memoria dedicada a cachear instrucciones.

Los micros RISC y EPIC tienen un tamaño de instrucciones estadístico medio notablemente superior al de los CISC, y además necesitan más instrucciones para procesar la misma cantidad de datos. Solucionan el problema mediante el uso de cachés de mayor tamaño.

17. Citar factores a favor y en contra que podrían influir para que el tamaño de las páginas fueran mayores que 4 kbytes.

A favor:

- Se harían menos conmutaciones de página.
- La administración de páginas sería más simple para el S.O., ya que habrían menos.
- El tamaño de las tablas de traducción sería menor.

En contra:

- Se desaprovecharía más memoria dentro de cada página.
- La fragmentación interna aumentaría.
- Forzaría a hacer intercambios con el disco duro con menor selectividad de los datos modificados.

18. Desarrollar una secuencia de las asignaciones de memoria que podrían haber ocurrido en un sistema para que el mapa de memoria física quede como aparece en la Figura 2-18.

1. En la carga del S.O., se alojan las dos primeras páginas para el Núcleo, que tienen una correspondencia directa con la memoria física. Además se reserva espacio para el caché de disco duro.
2. Se carga un proceso 3 ocupando las tres páginas siguientes: 2, 3, y 4.
3. Se carga la primera página del proceso 1 que se mapea en la página física 5. Ya que no accede por el momento a otras páginas, no se asignan más páginas físicas.
4. Se cargan las tres primeras páginas del proceso 2 en 6, 7, y 8.
5. El proceso 1 necesita acceder a las páginas lógicas 2 y 3 que son mapeadas en las páginas físicas 9 y 10.
6. Termina el proceso 3 dejando vacantes las tres páginas físicas 2, 3, y 4.

7. El proceso 2 accede a la memoria de su página 5, que es mapeada por el sistema en la página física 2.
8. El proceso 2 accede a sus páginas lógicas 4 y 6 que son mapeadas en las páginas físicas 3 y 4.
9. El proceso 2 requiere almacenar la página lógica 7 que es correspondida con la página física 11.
10. Transcurre el tiempo y todas las páginas de los procesos son accedidas excepto la 4 y la 6 del proceso 2, lo que las convierte en candidatas para ser llevadas a intercambio.
11. El proceso 1 necesita acceder a sus páginas 4 y 5. El sistema lleva las páginas físicas 3 y 4 a disco duro, para poder mapear las páginas lógicas del proceso 1 en las páginas físicas 3 y 4.