



**2º de Grado en Ingeniería Informática  
Sistemas Operativos**



## **TEMA 1 – INTRODUCCIÓN A LOS SISTEMAS OPERATIVOS**

### **Bibliografía**

El contenido de este documento se ha elaborado, principalmente, a partir de las siguientes referencias bibliográficas, y con propósito meramente académico y no lucrativo:

- W. Stallings. *Sistemas operativos, 5ª edición*. Prentice Hall, Madrid, 2005.
- A. S. Tanenbaum. *Sistemas operativos modernos, 3a edición*. Prentice Hall, Madrid, 2009.
- A. Silberschatz, G. Gagné, P. B. Galvin. *Fundamentos de sistemas operativos, séptima edición*. McGraw-Hill, 2005.
- A. McIver, I. M. Flynn. *Sistemas operativos, 6ª edición*. Cengage Learning, 2011.
- J. A. Alamansa, M. A. Canto Diaz, J. M. de la Cruz García, S. Dormido Bencomo, C. Mañoso Hierro. *Sistemas operativos, teoría y problemas*. Editorial Sanz y Torres, S.L, 2002.
- F. Pérez, J. Carretero, F. García. *Problemas de sistemas operativos: de la base al diseño, 2ª edición*. McGraw-Hill, 2003.
- S. Candela, C. Rubén, A. Quesada, F. J. Santana, J. M. Santos. *Fundamentos de Sistemas Operativos, teoría y ejercicios resueltos*. Paraninfo, 2005.
- J. Aranda, M. A. Canto, J. M. de la Cruz, S. Dormido, C. Mañoso. *Sistemas Operativos: Teoría y problemas*. Sanz y Torres S.L, 2002.
- J. Carretero, F. García, P. de Miguel, F. Pérez, *Sistemas Operativos: Una visión aplicada*. Mc Graw Hill, 2001.

## 1 ¿Qué es un sistema operativo?

Como definición básica utilizada durante muchos años, un sistema operativo es un programa en ejecución que administra el hardware de una computadora. Como definición más completa y actual, un sistema operativo es aquel programa que se ejecuta continuamente en una computadora, siendo todo lo demás, programas del sistema y programas de aplicación.

A lo largo de este tema, se hablará de **programa o proceso** indistintamente, siendo el el siguiente tema cuando se aclare y se asimile su diferencia. De forma genérica, un proceso es un programa cargado en memoria principal.

Recuerde que hay dos tipos de programas: 1) los **programas de sistema**, que se encargan de controlar las operaciones propias de la computadora, 2) los **programas de aplicación**, que resuelven problemas específicos a los usuarios (procesador de textos, navegador, etc). El sistema operativo, por tanto, sería un programa de sistema.

Actualmente, mas que dar una definición diciendo qué es un sistema operativo, lo más idóneo es exponer cuáles son las funciones u objetivos del mismo. El objetivo principal del sistema operativo es proporcionar una interfaz intermediaria entre la máquina desnuda o hardware y el usuario, de forma que se puedan gestionar los recursos de la computadora. De esta manera se da al usuario una visión de una máquina virtual con la que es factible comunicarse. Por tanto, el sistema operativo se encarga de ocultar la complejidad del hardware, proporcionando una interfaz más adecuada y sencilla para hacer un uso y gestión del sistema.

El sistema operativo, además actúa como un mediador, de forma que los programadores y los programas de aplicación puedan acceder y utilizar los recursos y servicios del sistema sin preocuparse de la gestión de los mismos.

Como cualquier otro programa en una computadora, el sistema operativo debe disponer de CPU para ejecutarse, de forma que el propio procesador no lo distingue del resto de procesos.

### 1.1 El sistema operativo como gestor/asignador de recursos

Una responsabilidad clave de los sistemas operativos es la gestión de los recursos disponibles en un sistema (espacio de memoria principal/secundaria/virtual, dispositivos de E/S, procesadores) y la planificación de su uso por parte de los distintos procesos activos. Podríamos poner el símil en el que un sistema operativo es similar a un gobierno. Como un gobierno, no realiza ninguna función útil por sí mismo: simplemente proporciona un entorno en el que otros programas pueden llevar a cabo un trabajo útil.

Al enfrentarse a numerosas y posiblemente conflictivas solicitudes de recursos, el sistema operativo debe decidir cómo asignar éstos a programas y usuarios específicos, de modo que la computadora pueda operar de forma eficiente y equitativa.

Cualquier asignación de recursos y política de planificación debe tener en cuenta tres factores:

- **Equitatividad.** Es deseable que todos los procesos que compiten por un determinado recurso se les conceda un acceso equitativo al mismo (evita inanición). Esto es especialmente cierto para trabajos de la misma categoría, es decir, trabajos con demandas

similares.

- **Respuesta diferencial.** Por otro lado, el sistema operativo puede necesitar discriminar entre diferentes clases de trabajos con diferentes requisitos de servicio. El sistema operativo debe tomar las decisiones de asignación y planificación con el objetivo de satisfacer el conjunto total de requisitos. Además, debe tomar las decisiones de forma dinámica. Por ejemplo, si un proceso está esperando por el uso de un dispositivo de E/S, el sistema operativo puede intentar planificar este proceso para su ejecución tan pronto como sea posible, a fin de liberar el dispositivo para posteriores demandas de otros procesos.
- **Eficiencia.** El sistema operativo debe intentar maximizar la productividad, minimizar el tiempo de respuesta, y, en caso de sistemas de tiempo compartido (aulas UCO, servidores de internet con aplicaciones a miles de clientes), acomodar tantos usuarios como sea posible. Estos criterios entran en conflicto, encontrar un compromiso adecuado en una situación particular es un problema objeto de investigación (técnicas matemáticas y heurísticas sobre optimización) sobre sistemas operativos. Adicionalmente, medir la actividad del sistema es importante para ser capaz de monitorizar el rendimiento y realizar los ajustes correspondientes.

La Figura 2.11 sugiere los principales elementos del sistema operativo relacionados con la planificación de procesos y la asignación de recursos en un entorno de multiprogramación. Es necesario una gestión adecuada de las llamadas al sistemas y las interrupciones por E/S para llevar a cabo los tres factores comentados anteriormente (estos términos se estudiarán mas adelante en mayor detalle).

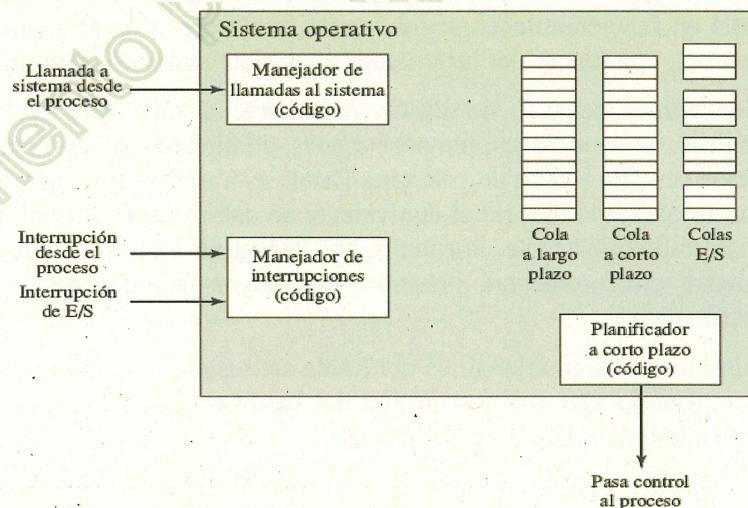


Figura 2.11. Elementos clave de un sistema operativo para la multiprogramación.

Como se puede ver en la figura, el sistema operativo mantiene un número de colas, cada una de las cuales es simplemente una lista de procesos esperando por algunos recursos. La cola a corto plazo está compuesta por procesos que se encuentran en memoria principal (o al menos una porción mínima esencial de cada uno de ellos está en memoria principal) y están listos para ejecutar, siempre que el procesador esté disponible. Cualquiera de estos procesos podría usar el procesador a

continuación.

La cola a largo plazo es una lista de nuevos trabajos esperando a utilizar el procesador. El sistema operativo añade trabajos al sistema transfiriendo un trabajo desde la cola a largo plazo hasta la cola a corto plazo. En este punto, se debe asignar una porción de memoria principal al proceso entrante. Por tanto, el sistema operativo debe estar seguro de que no sobrecarga la memoria o el tiempo de procesador, admitiendo demasiados procesos en el sistema.

Hay también una cola de E/S por cada dispositivo de E/S. Más de un proceso puede solicitar el uso del mismo dispositivo de E/S. Todos los procesos que esperan utilizar dicho dispositivo, se encuentran alineados en la cola del dispositivo. De nuevo, el sistema operativo debe determinar a qué proceso le asigna un dispositivo de E/S disponible.

Se verá de manera más específica este tipo de colas en el tema de planificación. Gran parte del esfuerzo en la investigación y desarrollo de los sistemas operativos ha sido dirigido a la creación de algoritmos de planificación y estructuras de datos que proporcionen **equitatividad, respuesta diferencial y eficiencia**.

## 1.2 *El sistema operativo como máquina virtual*

La utilización directa del hardware es difícil, especialmente para la realización de las operaciones de E/S. Por ejemplo, para operar directamente con el controlador de un CD-ROM hay que manejar del orden de 16 órdenes, y donde cada una puede tener más de una docena de parámetros, los cuales hay que empaquetar en un registro del dispositivo de 8 bytes, devolviéndose después de la operación un registro de 7 bytes con los estados y los campos de errores. A parte de eso, hay que verificar si el motor está en funcionamiento, si no lo está dar la orden de arranque, esperar a que gire a la velocidad adecuada, y entonces dar las órdenes de posicionamiento, lectura, escritura, etc.

El programador, por norma general, no desea enfrentarse a toda esta problemática (bits, transistores, puertas lógicas, interrupciones, temporizadores, administración de memoria, etc), sino que desea una abstracción sencilla y fácil de entender. Desde esta perspectiva, una de las funciones del sistema operativo es presentar al usuario el equivalente de una máquina virtual que es más fácil de programar que el hardware subyacente. Por tanto, una **máquina virtual** es aquella que, basada en una máquina hardware más elemental, presenta una mayor facilidad de uso de la misma incluyendo toda su funcionalidad.

Pues bien, para ocultar toda esta problemática del hardware, está el sistema operativo, de forma que el programador y el usuario ven una abstracción del hardware que se les presenta como una máquina virtual que entiende órdenes de nivel superior.

En este sentido, el sistema operativo tiene que proporcionar servicios para las funciones siguientes:

- **Creación de programas:** Existen otros programas del sistema, como son los depuradores, los editores, los enlazadores, los compiladores, que no son parte del sistema operativo, pero que son accesibles a través de él y que son necesarios para la creación de nuevos programas.
- **Ejecución de programas:** Para poder ejecutar un programa se tiene que realizar una serie de funciones previas, tales como cargar el código y los datos en la memoria principal, inicializar los dispositivos de E/S y preparar los recursos necesarios para la ejecución. Todo

esto lo gestiona el sistema operativo.

- **Operaciones de E/S:** Un proceso puede requerir una operación de E/S sobre un periférico. Pero cada uno tiene sus peculiaridades y un controlador específico con su conjunto de instrucciones. Como en el ejemplo del controlador del CD-ROM, es el propio sistema operativo el encargado de hacer todas esas funciones que permiten la lectura, escritura (en el caso de un CD-ROM re-grabable) y comunicación con los periféricos.
- **Manipulación y control del sistema de archivos:** Además de comunicarse con el controlador del periférico en donde está el sistema de archivos, el sistema operativo debe conocer la propia estructura y formato de almacenamiento del periférico, y proporcionar los mecanismos adecuados para su control y protección.
- **Detección de errores:** Hay una gran cantidad de errores, tanto del hardware como del software, que puede ocurrir. Por ejemplo, un mal funcionamiento de un periférico, fallos en la transmisión de los datos, errores de cálculo en un programa, divisiones por cero, fallos de memoria y de segmentación, violación de permisos, etc. El sistema operativo debe ser capaz de detectarlos y solucionarlos, o por lo menos hacer que tengan el menor impacto posible sobre el resto de las aplicaciones.
- **Control de acceso al sistema:** En sistemas de acceso compartido o en sistemas públicos, el sistema operativo debe controlar el acceso al mismo, vigilando quién tiene acceso y a qué recursos. Por este motivo tiene que tener mecanismos de protección de los recursos e implementar una adecuada política de seguridad, de forma que no pueda acceder quién no esté autorizado.
- **Elaboración de informes estadísticos:** Resulta muy conveniente conocer el grado de la utilización de los recursos, configuraciones y tiempo de respuesta. De esta forma se dispone de información que permite saber con antelación las necesidades futuras y configurar al sistema para dar el mejor rendimiento (veremos que este tipo de estadísticas pueden ser utilizadas incluso por el planificador del sistema).

## 2 Máquina multinivel o máquina virtual en capas

Los sistemas operativos puede dividirse en partes más pequeñas y más adecuadas que lo que permitían sistemas más antiguos como por ejemplo MS-DOS o UNIX. Con un **método de diseño modular arriba-abajo**, se determinan las características y la funcionalidad globales del sistema y se separan en componentes. De esta manera se puede mantener un control mucho mayor sobre la computadora y sobre las aplicaciones que hacen uso de dicha computadora, y los implementadores tiene más libertad para cambiar el funcionamiento interno del sistema.

Un sistema modular ofrece mayor libertad de cambio, menor esfuerzo de mantenimiento y ampliación, mayor facilidad de gestión y mayor facilidad de corrección de errores.

Desde el punto de vista de la programación, es importante ocultar los detalles a “ojos” de los niveles superiores, dado que deja libres a los programadores para implementar las rutinas de bajo nivel como prefieran, siempre que la interfaz externa de la rutina permanezca invariable y la propia rutina realice la tarea anunciada.

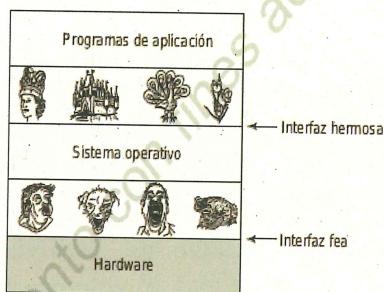


Figura 1-2. Los sistemas operativos ocultan el hardware feo con abstracciones hermosas.

Un sistema puede hacerse modular mediante una estructura en niveles, en el que el sistema operativo se divide en una serie de capas (niveles). El nivel inferior (nivel 0) es el hardware; el nivel superior (nivel N) es la interfaz de usuario. Esta estructura de niveles se ilustra en la Figura 2.12.

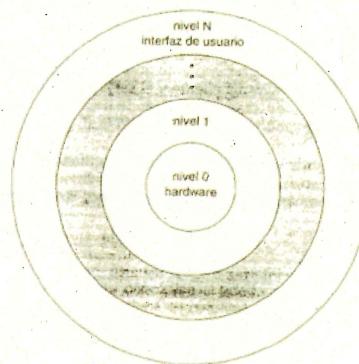


Figura 2.12 Un sistema operativo estructurado en niveles.

Un nivel de un sistema operativo es una implementación de un objeto abstracto formado por una serie de datos y por las operaciones que permiten manipular dichos datos (en la asignatura “Estructura de Datos” estudiará el concepto de objeto). Un nivel de un sistema operativo típico (por ejemplo, el nivel M) consta de estructuras de datos y de un conjunto de rutinas que los niveles superiores pueden invocar. A su vez, el nivel M puede invocar operaciones sobre los niveles inferiores. Surge entonces lo que llamamos **máquina multinivel**, que es una estructuración en capas bajo una serie de abstracciones, donde cada capa se apoya en la que está debajo de ella, y facilita el trabajo con el sistema operativo.

La principal ventaja del método de niveles es la simplicidad de construcción y depuración. Los niveles se seleccionan de modo que cada uno usa funciones (operaciones) y servicios de los niveles inferiores. Este método simplifica la depuración y la verificación del sistema. El primer nivel puede depurarse sin afectar al resto del sistema, dado que, por definición, sólo usa el hardware básico (que se supone correcto) para implementar sus funciones. Una vez que el primer nivel se ha depurado, puede suponerse correcto su funcionamiento mientras se depura el segundo nivel, etc. Si se encuentra un error durante la depuración de un determinado nivel, el error tendrá que estar localizado en dicho nivel, dado que los niveles inferiores a él ya se han depurado.

Cada nivel se implementa utilizando sólo las operaciones proporcionadas por los niveles inferiores. Un nivel no necesita saber cómo se implementan dichas operaciones; sólo necesita saber qué hacen esas operaciones. Por tanto, cada nivel oculta a los niveles superiores la existencia de determinadas estructuras de datos, operaciones y hardware.

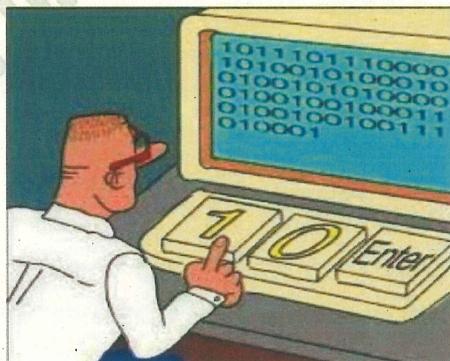
La principal dificultad con el método de niveles es la de definir apropiadamente los diferentes niveles. Dado que un nivel sólo puede usar los servicios de los niveles inferiores, es necesario realizar una planificación cuidadosa. Las implementaciones por niveles de manera excesiva tienden a ser poco eficientes. Por ejemplo, cuando un programa de usuario ejecuta una operación de E/S, realiza una llamada al sistema que será capturada por el nivel de E/S, el cual llamará al nivel de gestión de memoria, el cual a su vez llamará al nivel de planificación de la CPU, que pasará a continuación la llamada al hardware. En cada nivel, se pueden modificar los parámetros, puede ser necesario pasar datos, etc. Cada nivel añade así una carga de trabajo adicional a la llamada al sistema; el resultado neto es una llamada al sistema que tarda más en ejecutarse que en un sistema sin niveles.

Dicho esto, a continuación se expone como se puede diseñar un sistema en varios niveles para facilitar el uso y programación del hardware subyacente:

Las máquinas interpretan más fácilmente las señales *on* y *off*, lo que equivale a interpretar la presencia o la ausencia de voltaje. Así, el alfabeto de las máquinas está constituido por dos símbolos que son representados por 1 y 0 respectivamente. A cada uno de ellos se le conoce como dígito binario o bit, y sobre este alfabeto se construyen los comandos o instrucciones con los cuales nos comunicamos.

Las instrucciones que ejecutan los computadores son colecciones de bits que pueden ser vistos como números. Por ejemplo, el siguiente patrón de bits, podría indicar al computador que debe sumar dos números: 1000110010100000. El problema de esto es que escribir un programa a base de bits es algo tedioso y complicado para los programadores.

Imagínese haciendo sus programas de prácticas como se muestra en la siguiente figura...  
¿Impactante?



A continuación se muestra una representación de una máquina multinivel desde el punto de vista de un programador:

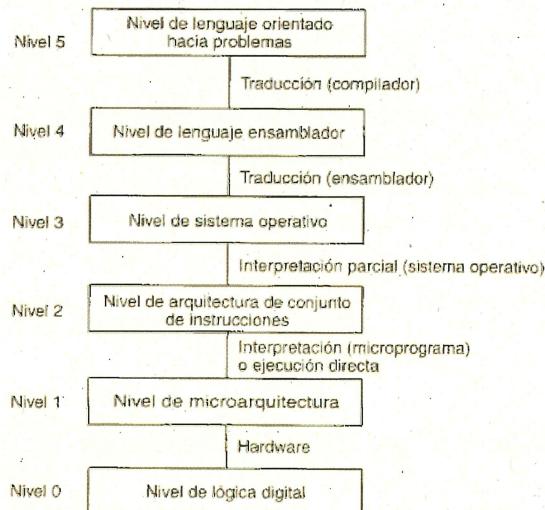
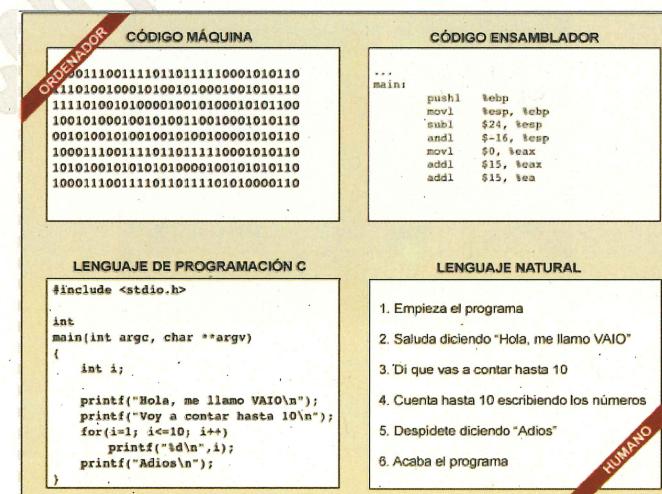


Figura 1-2. Computadora de seis niveles. El método de apoyo para cada nivel se indica inmediatamente abajo de él (junto con el nombre del programa de apoyo).

## 2.1 Lenguaje de alto nivel

Con el objetivo de expresar los programas en un lenguaje más cercano a la forma de pensamiento de los programadores surgen los **lenguajes de alto nivel (nivel 5)**. Para que los computadores puedan procesar estos nuevos tipos de lenguajes se desarrollaron otros programas que traducen programas escritos en un lenguaje de alto nivel a un lenguaje de más bajo nivel llamado **ensamblador (nivel 4)**, estamos hablando de los **compiladores**.

En la figura se muestra un ejemplo de un programa escrito en un lenguaje de alto nivel como C, que es traducido a un programa en lenguaje ensamblador por medio del compilador. Lea la figura de derecha a izquierda, empezando por la parte inferior.



## 2.2 Ensamblador

El lenguaje ensamblador todavía no es un lenguaje entendible por la máquina. Ensamblador es un tipo de programa informático, parecido a un compilador pero a más bajo nivel, que se encarga de traducir un fichero fuente escrito en **lenguaje ensamblador** (nivel 4) a un fichero objeto que contiene **código máquina (niveles 3 y 2)**. El código máquina está formado por un conjunto de instrucciones que determinan una serie de acciones que debe realizar la máquina. Por ejemplo, el programador escribe en lenguaje ensamblador “add A, B” (mnemónico) y el programa ensamblador traduce esta instrucción de más alto nivel en una instrucción “1000110010100000” de más bajo nivel (lenguaje de máquina). Según la arquitectura del microprocesador y el número de registros de éste, los mnemónicos y la traducción a código máquina difieren de unas a otras. Hay un ensamblador para cada tipo de procesador, pero hoy día todo está muy estandarizado en cuanto a arquitecturas. Los fabricantes de microprocesadores publican el repertorio de instrucciones en lenguaje ensamblador que sus máquinas pueden ejecutar.

En los niveles 3 y 2 de la figura de niveles, existen las mismas instrucciones máquina, la diferencia está en que en el nivel 3 el sistema operativo debe hacer determinadas reorganizaciones de las instrucciones a nivel de memoria, posible ejecución en paralelo de procesos, etc.

Veamos un ejemplo. En el lenguaje ensamblador para un procesador x86, la sentencia o mnemónico **MOV AL, 61h** asigna el valor hexadecimal 61 (97 decimal) al registro "AL". El programa ensamblador lee el mnemónico y produce su equivalente binario en lenguaje de máquina:

Binario: 1011000001100001

El código de máquina generado por el ensamblador consiste de 2 bytes, 10110000 01100001.

El primer byte contiene empaquetado la instrucción MOV y el código del registro hacia donde se va a mover el dato. En el segundo byte se especifica el número 61h, escrito en binario como 01100001, que se asignará al registro AL, quedando la sentencia ejecutable como:

|        |      |   |
|--------|------|---|
| 1011   | 0000 | 01100001  |
|        |      |   |
|        |      | +---- Número 97 en decimal (61h en hexadecimal) |
|        | +--  | Registro AL                                     |
| +----- |      | Instrucción MOV                                 |

Cabe decir aquí, que el funcionamiento del sistema operativo, es decir, el propio código del mismo, las funciones que lo rigen y el cómo trabaja, está codificado a nivel de instrucciones en código máquina. El código máquina del sistema operativo se carga al arrancar nuestra computadora para así poder proporcionar servicios y gestionar recursos para el usuario. A este código codificado en instrucciones máquina no se puede acceder por parte de un usuario normal (se ejecuta, como veremos, en modo *kernel* o núcleo), el cual solo puede hacer cambios en el nivel 5 o como mucho en el nivel 4, siendo el diseñador del sistema operativo el que se encarga de hacer modificaciones a más bajo nivel. Esto da lugar a lo que se llama modo usuario y modo núcleo (*kernel*), de lo cual se hablará posteriormente.

## 2.3 Microprogramación

El diseño de microprocesadores de propósito general conoce dos técnicas que conducen a una clasificación de éstos en dos grupos:

- Los microprocesadores "cableados": aquellos que tienen una unidad de control específicamente diseñada sobre el silicio, para un juego de instrucciones concreto. En este caso el hardware se encarga directamente de ejecutar las microinstrucciones del nivel 2.
- Los microprocesadores "microprogramados": aquellos que tienen una unidad de control genérica o prediseñada, y que implementan un juego de instrucciones u otro dependiendo de un microprograma software (paso de nivel 2 a nivel 1).

En las máquinas en las que no existe el nivel de microprogramación, las instrucciones del nivel de máquina son realizadas directamente por los circuitos electrónicos (el hardware, el nivel 0), es decir, se pasa del nivel 2 al nivel 0, sin pasar por el 1. Hoy día la microprogramación ha desaparecido prácticamente por completo. Esto se debe a los siguientes factores:

- Ya existen herramientas avanzadas para diseñar complejas unidades de control con millones de transistores litografiados. Estas herramientas prácticamente garantizan la ausencia de errores de diseño.
- Las unidades de control cableadas tienen un rendimiento significativamente mayor que cualquier unidad microprogramada, resultando más competitivas.

En el caso de que un sistema necesite ser microprogramado, existe otro paso intermedio (**del nivel 2 al nivel 1**) antes de llegar al hardware (nivel 0). El lenguaje máquina no es el que se pondría directamente sobre la lógica digital (circuitos integrados, transistores, puertas lógicas, etc) o nivel 0, sino que se interpretaría del nivel 2 al nivel 1 por un programa llamado **microprograma**. El microprograma transforma el lenguaje máquina en microinstrucciones con un determinado formato de unos y ceros, adaptando instrucciones de un propósito más general a microinstrucciones específicas que dependen de la arquitectura (una instrucción en lenguaje máquina se “parte” en tareas más sencillas que dar lugar a varias microinstrucciones). Estas microinstrucciones ya si se pueden ejecutar en el hardware interpretadas por el microprograma y siguiendo un determinado flujo o trayectoria de datos hasta la unidad aritmético-lógica (ALU) del procesador. Por tanto, la microprogramación constituye un intérprete entre el lenguaje máquina del nivel 2 y el hardware del microprocesador (nivel 0) que dice qué camino deben seguir las microinstrucciones en el procesador y sus registros.

En las máquinas que no están microprogramadas, esa trayectoria de datos la controla directamente el hardware. El cómo se hace de manera concreta es algo que queda fuera de esta asignatura y se enfoca más a temas de arquitectura de computadores.

### 3 Elementos básicos y organización de un computador

Al más alto nivel, conocido como arquitectura de *Von Neumann*, un computador digital electrónico consta de:

- Una **unidad de procesamiento** (*Central Processing Unit, CPU*), que contiene una unidad aritmético lógica, un registro de instrucciones, un contador de programa y otra serie de registros auxiliares. La CPU controla el funcionamiento del computador y realiza sus funciones de procesamiento de datos.
- Una memoria para almacenar tanto datos como instrucciones, conocida como **memoria principal**. Esta memoria es volátil, es decir, cuando se apaga el computador, se pierde su contenido. En contraste, el contenido de la memoria del disco se mantiene incluso cuando se apaga el computador. A la memoria principal se le denomina también memoria real o memoria primaria.
- **Módulos o controladoras de entrada y salida (E/S)**. Transfieren los datos entre el computador y su entorno externo. El entorno externo está formado por diversos dispositivos, incluyendo dispositivos de memoria secundaria (discos duros, CD-ROMS, pendrives, etc), equipos de comunicaciones y terminales.

Estos componentes se interconectan de manera que se pueda lograr la función principal del computador, que es ejecutar programas. Para ello se usan los **buses del sistema**, que proporcionan comunicación entre los procesadores, la memoria principal y los módulos de E/S. La Figura 1.1 muestra estos componentes de más alto nivel.

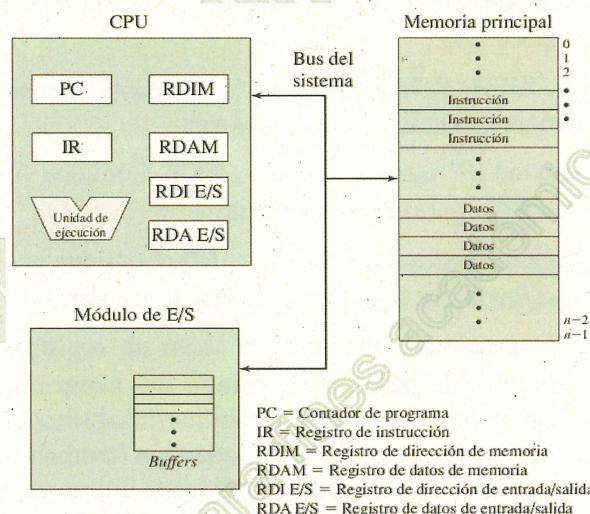


Figura 1.1. Componentes de un computador: visión al más alto nivel.

Un módulo o controladora de E/S transfiere datos desde los dispositivos externos hacia el procesador y la memoria, y viceversa. Contiene *buffers* (zonas de almacenamiento internas) que mantienen temporalmente los datos hasta que se puedan enviar. Cada controladora de dispositivo se encarga de un tipo específico de dispositivo, por ejemplo, unidades de disco, dispositivos de audio y pantallas de vídeo. La CPU y las controladoras de dispositivos pueden funcionar de forma

concurrente, compitiendo por la memoria principal. Para asegurar el acceso de forma ordenada a la memoria principal, se proporciona una controladora de memoria cuya función es sincronizar el acceso a la misma.

Un módulo de memoria consta de un conjunto de posiciones definidas mediante direcciones numeradas secuencialmente. Cada posición contiene un patrón de bits que se puede interpretar como una instrucción o como datos.

### 3.1 El procesador y sus registros

Un procesador incluye un conjunto de registros que proporcionan un tipo de memoria que es más rápida y de menor capacidad que la memoria principal. Una de las funciones del procesador es el intercambio de datos con la memoria y los dispositivos de E/S. Para este fin, se utilizan normalmente una serie registros internos al procesador (todo se estudia en más profundidad en materias de arquitectura y tecnología de computadores):

- Un **registro de dirección de memoria (RDIM)**, que especifica la dirección de memoria de la siguiente lectura o escritura.
- Un **registro de datos de memoria (RDAM)**, que contiene los datos que se van a escribir en la memoria o recibe los datos leídos de la memoria.
- Un **registro de dirección de E/S (RDIE/S)**, que especifica un determinado dispositivo de E/S.
- Un **registro de datos de E/S (RDAE/S)**, que permite el intercambio de datos entre un módulo de E/S y el procesador.

Además de los registros RDIRM, RDAM, RDIE/S y RDAE/S mencionados anteriormente, los siguientes son esenciales para la ejecución de instrucciones:

- **Contador de programa (Program Counter, PC)**. Contiene la dirección de la próxima instrucción que se leerá (cargará) de la memoria.
- **Registro de instrucción (Instruction Register, IR)**. Contiene la última instrucción leída, es decir, la instrucción cargada que hay que comenzar a ejecutar.
- Todos los diseños de procesador incluyen también un registro, o conjunto de registros, conocido usualmente como la **palabra de estado del programa (Program Status Word, PSW)**. El registro PSW contiene un bit para habilitar/inhabilitar las interrupciones, un bit de modo usuario/supervisor y bits de códigos de condición (también llamados indicadores), que son bits cuyo valor lo asigna normalmente el hardware del procesador teniendo en cuenta el resultado de operaciones. Por ejemplo, una operación aritmética puede producir un resultado positivo, negativo, cero o desbordamiento. Además de almacenarse el resultado en sí mismo en un registro o en la memoria, se fija también un código de condición en concordancia con el resultado de la ejecución de la instrucción aritmética. Posteriormente, se puede comprobar el código de condición como parte, por ejemplo, de una operación de salto condicional.

Se puede hablar también de dos tipos de registros según sean visibles al usuario en un lenguaje de programación o no:

- **Registros visibles para el usuario.** Permiten al programador en lenguaje máquina o en ensamblador, minimizar las referencias a memoria principal, optimizando así el uso de registros. Para lenguajes de alto nivel, un compilador que realice optimización intentará tomar decisiones inteligentes sobre qué variables se asignan a registros y cuáles a posiciones de memoria principal. Algunos lenguajes de alto nivel, tales como C, permiten al programador sugerir al compilador qué variables deberían almacenarse en registros. Dependiendo de la arquitectura del procesador, estos registros pueden cambiar. Los tipos de registros que están normalmente disponibles son registros de datos (por ejemplo, una operación de suma) o de dirección (contienen direcciones de memoria principal de datos e instrucciones).
- **Registros de control y estado (no visibles para el usuario).** Usados por el procesador para controlar su operación y por rutinas privilegiadas del sistema operativo para controlar la ejecución de programas. Se puede acceder mediante instrucciones de máquina ejecutadas en lo que se denomina modo de control, *kernel* o núcleo (se estudiará más adelante). Por supuesto, diferentes máquinas tendrán distintas organizaciones de registros y utilizarán diferente terminología.

### 3.2 Estructura de almacenamiento

Los programas de la computadora deben hallarse en la memoria principal (también llamada memoria RAM, *random-access memory* o memoria de acceso aleatorio) para ser ejecutados. La memoria principal es el único área de almacenamiento de gran tamaño (millones o miles de millones de bytes) a la que el procesador puede acceder directamente. Habitualmente, se implementa con una tecnología de semiconductores que forman una matriz de palabras de memoria. Cada palabra tiene su propia dirección. La interacción se consigue a través de una secuencia de carga (*load*) o almacenamiento (*store*) de instrucciones en direcciones específicas de memoria. La instrucción *load* mueve una palabra desde la memoria principal a un registro interno de la CPU, mientras que la instrucción *store* mueve el contenido de un registro a la memoria principal. Aparte de las cargas y almacenamientos explícitos, la CPU carga automáticamente instrucciones desde la memoria principal para su ejecución.

Como veremos en la siguiente sección, en un ciclo típico instrucción-ejecución primero se extrae una instrucción de memoria y se almacena dicha instrucción en el registro de instrucciones IR. A continuación, la instrucción se decodifica y puede dar lugar a que se extraigan operandos de la memoria y se almacenen en algún registro interno. Después de ejecutar la instrucción con los necesarios operandos, el resultado se almacena de nuevo en memoria.

Idealmente, es deseable que los programas y los datos residan en la memoria principal de forma permanente. Usualmente, esta situación no es posible por las dos razones siguientes:

- Normalmente, la memoria principal es demasiado pequeña como para almacenar todos los programas y datos necesarios de forma permanente.
- La memoria principal es un dispositivo de almacenamiento volátil que pierde su contenido

cuando se quita la alimentación.

Por tanto, la mayor parte de los sistemas informáticos proporcionan almacenamiento secundario como una extensión de la memoria principal. El requerimiento fundamental de este almacenamiento secundario es que se tienen que poder almacenar grandes cantidades de datos de forma permanente. El dispositivo de almacenamiento secundario más común es el disco magnético, actualmente también los disco de estado sólido o SSD. Estos discos proporcionan un sistema de almacenamiento tanto para programas como para datos. La mayoría de los programas (exploradores web, compiladores, procesadores de texto, hojas de cálculo, etc.) se almacenan en un disco hasta que se cargan en memoria.

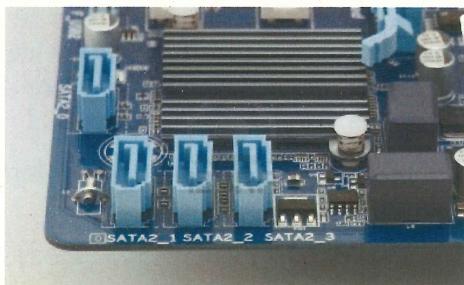
Otros sistemas de almacenamiento son la memoria caché, los CD-ROM, cintas magnéticas, etc. Cada sistema de almacenamiento proporciona las funciones básicas para guardar datos y mantener dichos datos hasta que sean recuperados en un instante posterior. Las principales diferencias entre los distintos sistemas de almacenamiento están relacionadas con la velocidad, el coste, el tamaño y la volatilidad.

### 3.3 Estructura de Entrada/Salida (E/S)

Los de almacenamiento son sólo uno de los muchos tipos de dispositivos de E/S que hay en un sistema informático. Gran parte del código del sistema operativo se dedica a gestionar la entrada y la salida, debido a su importancia para la fiabilidad y rendimiento del sistema y debido también a la variada naturaleza de los dispositivos. Vamos a comentar de manera introductoria los conceptos de la E/S.

Una computadora de propósito general consta de una o más CPUs y de múltiples controladoras de dispositivo que se conectan a través de un bus común. Cada controladora de dispositivo se encarga de un tipo específico de dispositivo. Dependiendo de la controladora, puede haber más de un tipo de dispositivo conectado a ella. Por ejemplo, dos o más dispositivos pueden estar conectados a una controladora SATA (*Serial Advanced Technology Attachment*) o a una controladora USB (*Universal Serial Bus*).

Una controladora de dispositivo mantiene algunos búferes locales y un conjunto de registros de propósito especial. La controladora del dispositivo es responsable de transferir los datos entre los dispositivos periféricos que controla y su búfer local. Un controlador/a puede estar integrado o no en placa base.



Los dispositivos deberán incorporar en su electrónica un interfaz para trabajar con una controladora concreta, o al menos disponer de un conector de adaptación.



Normalmente, los sistemas operativos tienen un **controlador/a de dispositivo software** (llamado *driver*) para cada **controlador/a de dispositivo hardware** (llamado *controller*). Es muy importante no confundir la **controlador/a de dispositivo hardware** con la **controlador/a de dispositivo software**. Aunque para ambos se utiliza asiduamente el término "controlador/a" el primero es físico (electrónico) y el segundo es un componente software que se integra con el Sistema Operativo para comunicarse con el correspondiente dispositivo. Este software controlador del dispositivo es capaz de entenderse con la controladora hardware y presenta al resto del sistema operativo una interfaz uniforme mediante la cual comunicarse con el dispositivo.

Al iniciar una operación de E/S, el *driver* carga los registros apropiados de la controladora hardware. Ésta, a su vez, examina el contenido de estos registros para determinar qué acción realizar (como, por ejemplo, "leer un carácter del teclado"). La controladora inicia entonces la transferencia de datos desde el dispositivo a su búfer local. Una vez completada la transferencia de datos, la controladora informa al *driver*, a través de una interrupción, de que ha terminado la operación. El *driver* devuelve entonces el control al sistema operativo, devolviendo posiblemente los datos, o un puntero a los datos, si la operación ha sido una lectura. Para otras operaciones, el *driver* devuelve información de estado.

Esta forma de E/S controlada por interrupción resulta adecuada para transferir cantidades pequeñas de datos, pero representa un desperdicio de capacidad de procesamiento cuando se usa para movimientos masivos de datos, como en la E/S de disco. Para resolver este problema, se usa el acceso directo a memoria (DMA, *direct memory access*). Esto se estudiará más detenidamente, pero de manera resumida, después de configurar búferes, punteros y contadores para el dispositivo de E/S, la controladora hardware transfiere un bloque entero de datos entre su propio búfer y la memoria, sin que intervenga la CPU. Sólo se genera una interrupción por cada bloque, para decir al controlador software del dispositivo que la operación se ha completado, en lugar de la interrupción por byte generada en los dispositivos de baja velocidad. Mientras la controladora hardware realiza estas operaciones, la CPU está disponible para llevar a cabo otros trabajos.

## 4 Ejecución de instrucciones

Un programa que va a ejecutarse en un procesador consta de un conjunto de instrucciones almacenado en memoria principal. En su forma más simple, el procesamiento de una instrucción consta de dos pasos: el procesador lee, busca instrucciones de la memoria, una cada vez, y ejecuta cada una de ellas. La ejecución del programa consiste en repetir el proceso de búsqueda y ejecución de instrucciones. La ejecución de la instrucción puede involucrar varias operaciones dependiendo de la naturaleza de la misma.

Se denomina **ciclo de instrucción** al procesamiento requerido por una única instrucción. En la Figura 1.2 se describe el ciclo de instrucción utilizando la descripción simplificada de dos pasos. A estos dos pasos se les denomina **fase de búsqueda** y **de ejecución**. La ejecución del proceso se detiene sólo si se apaga la máquina, se produce algún tipo de error irrecuperable o se ejecuta una instrucción del proceso que para el procesador.

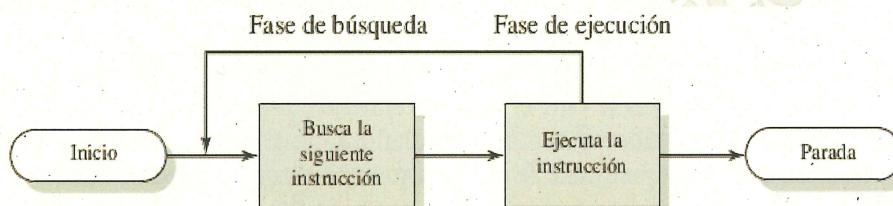


Figura 1.2. Ciclo de instrucción básico.

### 4.1 Búsqueda y ejecución de una instrucción

Al principio de cada ciclo de instrucción, el procesador lee una instrucción de la memoria. En un procesador típico, el contador del programa (PC) almacena la dirección de la siguiente instrucción que se va a leer. La instrucción leída se carga dentro de un registro del procesador conocido como registro de instrucción (IR). A menos que se le indique otra cosa, el procesador siempre incrementa el PC después de cada instrucción cargada en el IR, de manera que se leerá la siguiente instrucción en orden secuencial, es decir, la instrucción situada en la siguiente dirección de memoria más alta.

Considere, por ejemplo, un computador simplificado en el que cada instrucción ocupa una palabra de memoria de 16 bits. Suponga que el contador del programa está situado en la posición 300. El procesador leerá la instrucción de la posición 300. En sucesivos ciclos de instrucción completados satisfactoriamente, se leerán instrucciones de las posiciones 301, 302, 303, y así sucesivamente. Esta secuencia se puede alterar con instrucciones de tipo salto.

La instrucción contiene bits que especifican la acción que debe realizar el procesador (parte de un mnemónico en lenguaje ensamblador). El procesador interpreta la instrucción y lleva a cabo la acción requerida. En general, las acciones que se pueden realizar con una instrucción se dividen en cuatro categorías:

- **Procesador-memoria.** Se pueden transferir datos desde el procesador a la memoria o viceversa.

- **Procesador-E/S.** Se pueden enviar datos a un dispositivo periférico o recibirlas desde el mismo, transfiriéndolas entre el procesador y un módulo de E/S.
- **Procesamiento de datos.** El procesador puede realizar algunas operaciones aritméticas o lógicas sobre los datos.
- **Control.** Una instrucción puede especificar que se va a alterar la secuencia de ejecución. Por ejemplo, el procesador puede leer una instrucción de la posición 149, que especifica que la siguiente instrucción estará en la posición 182. El procesador almacenará en el contador del programa un valor de 182. Como consecuencia, en la siguiente fase de búsqueda, se leerá la instrucción de la posición 182 en vez de la 150.

## 4.2 Ejemplo de ejecución de instrucciones

Considere un ejemplo sencillo utilizando una máquina hipotética que incluye las características mostradas en la Figura 1.3.

|              |   |           |    |
|--------------|---|-----------|----|
| 0            | 3 | 4         | 15 |
| Código-de-op |   | Dirección |    |

(a) Formato de instrucción

|   |   |          |
|---|---|----------|
| 0 | 1 | 15       |
| S |   | Magnitud |

(b) Formato de un entero

Contador de programa (PC) = Dirección de la instrucción  
 Registro de instrucción (IR) = Instrucción que se está ejecutando  
 Acumulador (AC) = Almacenamiento temporal

(c) Registros internos de la CPU

0001 = Carga AC desde la memoria  
 0010 = Almacena AC en memoria  
 0101 = Suma a AC de la memoria

(d) Lista parcial de códigos-de-op.

Figura 1.3. Características de una máquina hipotética.

El procesador contiene un único registro de datos, llamado el acumulador (AC). Tanto las instrucciones como los datos tienen una longitud de 16 bits, estando la memoria organizada como una secuencia de palabras de 16 bits. El formato de la instrucción proporciona 4 bits para el código de operación, permitiendo hasta  $2^4 = 16$  códigos de operación diferentes (representados por un único dígito hexadecimal 3). Con los 12 bits restantes del formato de la instrucción, se pueden direccionar directamente hasta  $2^{12} = 4.096$  (4K) palabras de memoria (denotadas por tres dígitos hexadecimales).

La Figura 1.4 ilustra una ejecución parcial de un programa, mostrando las partes relevantes de la memoria y de los registros del procesador. El fragmento de programa mostrado suma el contenido de la palabra de memoria en la dirección 940 al de la palabra de memoria en la dirección 941, almacenando el resultado en esta última posición. Se requieren tres instrucciones, que corresponden a tres fases de búsqueda y de ejecución (1-2; 3-4; 5-6), como se describe a continuación:

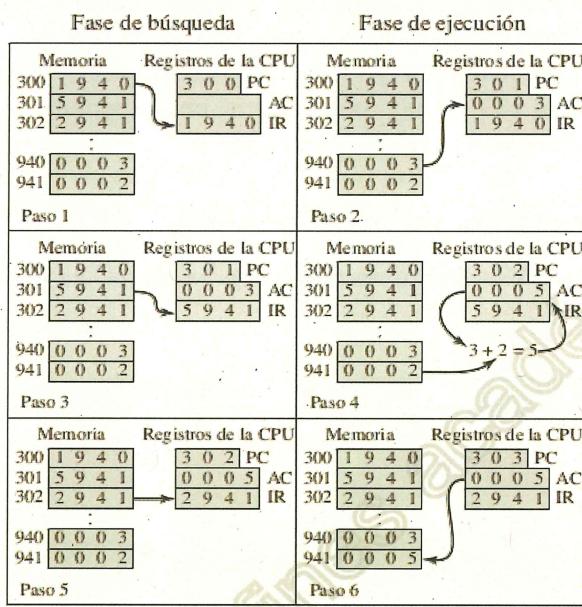


Figura 1.4. Ejemplo de ejecución de un programa (contenido de la memoria y los registros en hexadecimal).

1. El PC contiene el valor 300, la dirección de la primera instrucción. Esta instrucción (el valor 1940 en hexadecimal) se carga dentro del registro de instrucción IR y se incrementa el PC. Note que este proceso involucra el uso del registro de dirección de memoria (RDIM) y el registro de datos de memoria (RDAM). Para simplificar, no se muestran estos registros intermedios.
2. Los primeros 4 bits (primer dígito hexadecimal) en el IR indican que en el AC se va a cargar un valor leído de la memoria. Los restantes 12 bits (tres dígitos hexadecimales) especifican la dirección de memoria, que es 940.
3. Se lee la siguiente instrucción (5941) de la posición 301 y se incrementa el PC.
4. El contenido previo del AC y el contenido de la posición 941 se suman y el resultado se almacena en el AC.
5. Se lee la siguiente instrucción (2941) de la posición 302 y se incrementa el PC.
6. Se almacena el contenido del AC en la posición 941.

En este ejemplo se necesitan tres ciclos de instrucción, y cada uno consta de una fase de búsqueda y una fase de ejecución para sumar el contenido de la posición 940 al contenido de la 941. Con un juego de instrucciones más complejos, se necesitarían menos ciclos de instrucción. La mayoría de los procesadores modernos incluyen instrucciones que contienen más de una dirección.

Por tanto, la fase de ejecución de una determinada instrucción puede involucrar más de una referencia a memoria. Asimismo, en vez de referencias a memoria, una instrucción puede especificar una operación de E/S. Mientras más bits tenga una instrucción (32 o 64 bits) más operaciones sobre memoria se pueden hacer, y mientras más megahercios (Mhz) tenga un procesador más instrucciones por segundo se podrán buscar-ejecutar.

## 5 Interrupciones y manejador de interrupciones

Mientras el procesador está realizando fases de búsqueda-ejecución de instrucciones, éste puede verse interrumpido por un suceso o evento. La ocurrencia de un suceso normalmente se indica mediante una interrupción bien hardware o bien software.

Los tipos de **interrupciones hardware**, las cuales se pueden activar en cualquier instante enviando una señal a la CPU a través de un bus del sistema, son:

- **Interrupciones de E/S:** Generada por un controlador/a de dispositivos de entrada-salida para señalar la conclusión normal de una operación o para indicar diversas condiciones de error.
- **Interrupciones por fallo de hardware:** Se pueden producir por cortes en el suministro de energía o zonas corruptas de memoria.

Los tipos de **interrupciones software**, también imprevisibles, son:

- **Interrupciones de programa:** Generadas por alguna condición que se produce como resultado de la ejecución de una instrucción: desbordamiento aritmético, división por cero, intento de ejecución de instrucciones máquina ilegales, referencias a espacios de memoria no permitidos o debidamente reservados, etc. Se les suele llamar también **excepciones**.
- **Interrupciones por temporizador:** Se pueden generar por un temporizador del procesador, y permiten al sistema operativo realizar ciertas funciones de forma regular. Cuando se estudie el tema de planificación, se verá que una interrupción puede venir cuando un proceso ya ha agotado su cuota de CPU, de forma que hay que dejar paso a otro que esté listo para ejecutarse.

De manera general, cuando se interrumpe a la CPU, ésta deja lo que está haciendo e inmediatamente transfiere la ejecución a una posición de memoria principal fijada y establecida. Normalmente, dicha posición contiene la dirección de inicio donde se encuentra una **rutina de servicio a la interrupción (Interrupt Service Routine – ISR)**. La ISR se ejecuta, trata la interrupción y, cuando ha terminado, la CPU reanuda la operación que estuviera haciendo.

Cada diseño de computadora tiene su propio mecanismo de interrupciones, aunque hay algunas funciones comunes, y es que la interrupción debe transferir el control a la ISR apropiada. El método más simple para llevar a cabo esta idea es que la transferencia consista en invocar a una rutina ISR genérica para examinar la información de la interrupción; esa **rutina genérica**, a su vez, debe llamar a la rutina específica de tratamiento de la interrupción según cuál sea el origen que la produce. Sin embargo, las interrupciones deben tratarse rápidamente y este método es algo lento. En consecuencia, como sólo es posible un número predefinido de interrupciones, puede utilizarse otro sistema, consistente en disponer de una **tabla de punteros a las diferentes rutinas de**

**interrupción** (se le suele llamar **vector de interrupciones**), con el fin de proporcionar la velocidad necesaria. Concretamente, un vector de interrupciones es una tabla o matriz, ubicada en memoria principal, con estructura de vector, y en donde cada elemento contiene la dirección de memoria de las diferentes rutinas ISR que se pueden ejecutar. Esta tabla suele ocupar las posiciones más bajas de memoria. El vector de interrupciones se indexa mediante un número de interrupción único, que se proporciona en la propia solicitud de interrupción, por medio de una controladora, y sirve para obtener la dirección de la ISR específica para el tratamiento de la interrupción concreta que se ha producido. Es decir, en la propia interrupción, ya podría ir implícito el índice del vector de interrupciones que se debe cargar para ejecutar una ISR específica. Sistemas operativos tan diferentes como Windows y UNIX manejan las interrupciones de este modo.

Es muy importante saber que cuando se invoca a una ISR se almacena la dirección de la instrucción interrumpida correspondiente al proceso actual, es decir, la siguiente instrucción que se iba a ejecutar cuando se produjo la interrupción, además de otros valores de los registros del procesador, ya que la propia rutina ISR puede utilizar registros del procesador que guardan información sobre el proceso que se estaba ejecutando en ese momento. Después de atender a la interrupción, la dirección de retorno guardada se carga en el contador de programa, y los registros salvados vuelven a cargar, de manera que el cálculo interrumpido del programa que estaba en ejecución se reanuda, como si la interrupción no se hubiera producido. Se detallará más adelante, pero este paso se llama “salvado y restauración de contexto” de un proceso.

Mientras que actúa una interrupción, estas están inhabilitadas. Sin embargo, esto no se puede realizar en sistemas donde determinadas interrupciones requieren una respuesta rápida. Por ello, surge la noción de prioridad entre las interrupciones, de forma que una interrupción puede ser interrumpida si llega otra con una prioridad más alta, por lo que solo se inhabilitarían las interrupciones de igual o menor prioridad. Se verán ejemplos de interrupciones múltiples en las siguientes secciones

Note que cuando estamos hablando de interrupción la “dirección” va hacia el procesador, es decir, el procesador está realizando una tarea concreta y de pronto se ve interrumpido, ya sea por su propia tarea (división por cero por ejemplo) o por otro tipo de suceso (por ejemplo, un controlador de disco duro, diciendo que ya tiene disponible para transferir a memoria un fichero solicitado).

Otra cosa es invocar una operación de E/S, que resulta en una “llamada al sistema” (se estudiará posteriormente), pero eso NO ES UNA INTERRUPCIÓN.

## 5.1 Adición de la fase de interrupción

Desde el punto de vista de las interrupciones, el tratamiento de las interrupciones hardware de E/S son tremadamente importantes. La mayoría de los dispositivos de E/S son mucho más lentos que el procesador.

Suponga un proceso en el que el procesador está transfiriendo datos a una impresora utilizando el esquema de ciclo de instrucción de la Figura 1.2.

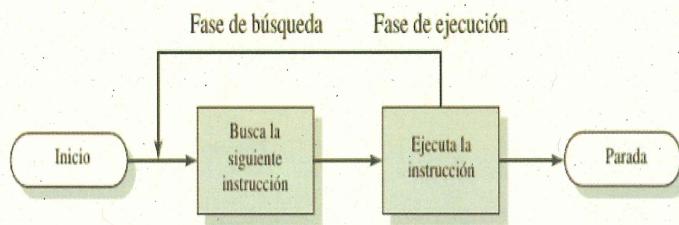
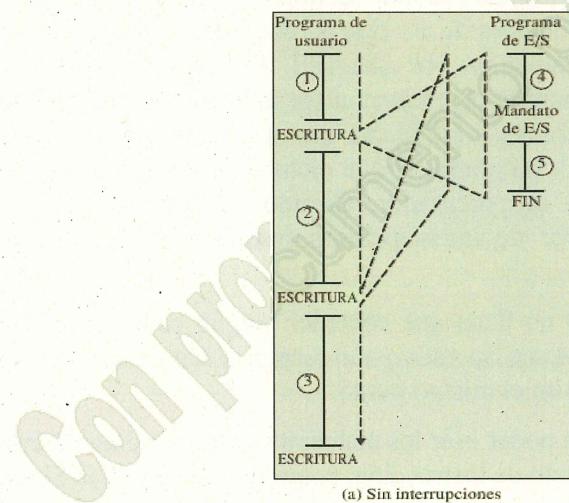


Figura 1.2. Ciclo de instrucción básico.

Después de cada petición de escritura a la impresora, el procesador debe parar y permanecer inactivo hasta que la impresora lleve a cabo su labor (vacíe su *buffer* local e imprima). La longitud de esta pausa puede ser del orden de muchos miles o incluso millones de ciclos de instrucción. Claramente, es un enorme desperdicio de la capacidad del procesador. Para dar un ejemplo concreto, considere un computador personal que operase a 1GHz, lo que le permitiría ejecutar aproximadamente  $10^9$  instrucciones por segundo. Un típico disco duro tiene una velocidad de rotación de 7200 revoluciones por minuto, que corresponde con un tiempo de rotación de media pista de 4 ms., que es 4 millones de veces más lento que el procesador.

Continuando con el programa anterior, suponga que éste realiza una serie de peticiones de escritura intercaladas con el procesamiento del resto del código del programa (ver Figura).



(a) Sin interrupciones

El círculo 4 consiste en preparar la operación de E/S, es decir comprobar si la impresora está disponible o no, e invocar las rutinas necesarias para el envío de la información a sus *buffers* internos (veremos que esto lo puede hacer un controlador/a de E/S). El círculo 5 es la operación en sí, es decir, que la impresora imprima en papel la información que tiene en su *buffer* e indique al sistema operativo que todo ha ido correcto o que ha habido algún error.

Debido a que la operación de impresión puede tardar un tiempo relativamente largo hasta que se completa, el procesador se queda esperando a que se termine; por ello, el programa de usuario se detiene en la llamada de ESCRITURA durante un periodo de tiempo considerable, “desperdiéndose” el uso de la CPU hasta que se haya impreso en el papel.

Pues bien, en vez de que el procesador se quede a la espera de que se lleve a cabo la operación de impresión, lo que se va a hacer es que éste continúe haciendo otras cosas (ver Figuras).

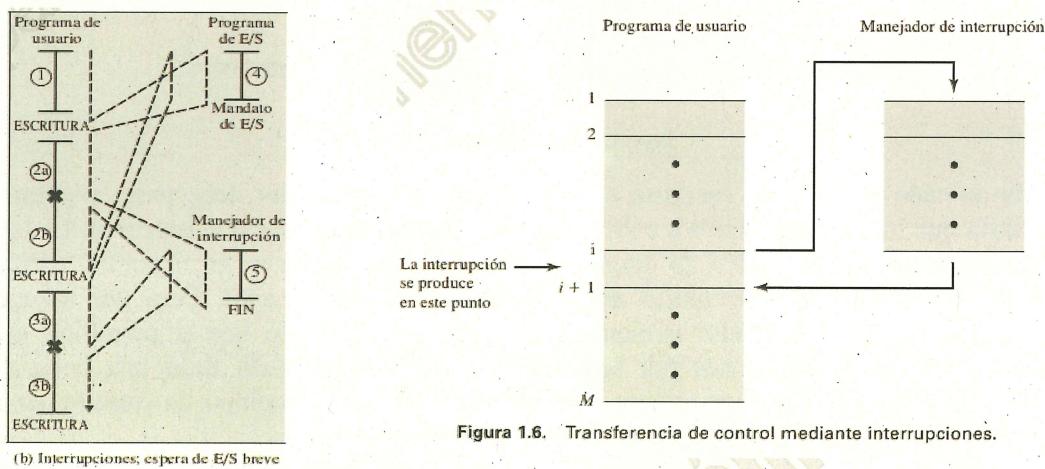


Figura 1.6. Transferencia de control mediante interrupciones.

En este ejemplo y siguiendo esta última idea, se continuará con la siguiente línea de código del programa que se está ejecutando. Entonces, cuando la impresora haya terminado de imprimir, lo indica mandando a través de la controladora de ese tipo de dispositivo, una petición de interrupción al sistema operativo para que sea atendida por una ISR. De cara al programa de usuario, la interrupción suspende la secuencia normal de ejecución (se indican con una “X” los puntos en los que se produce cada interrupción). Téngase en cuenta que se puede producir una interrupción en cualquier punto de la ejecución de un programa, y que dicha interrupción se produce en dirección desde el dispositivo externo hacia el procesador. Cuando se completa el tratamiento de la interrupción por parte del manejador de interrupciones (rutina IRS), de nuevo se reanuda la ejecución por donde iba.

Por tanto, el programa de usuario no tiene que contener ningún código especial para tratar las interrupciones, el procesador y el sistema operativo son responsables de suspender el programa de usuario y, posteriormente, reanudarlo en el mismo punto.

Dicho esto falta algo, y es que para poder usar los conceptos anteriormente descritos, es necesario añadir una **fase de interrupción** al ciclo de instrucción, como se muestra en la Figura 1.7.

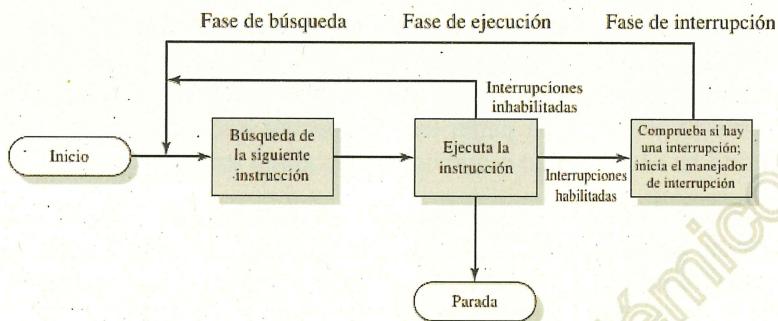


Figura 1.7. Ciclo de instrucción con interrupciones.

Por consiguiente, y resumiendo teniendo en cuenta la figura anterior, en la fase de interrupción, el procesador comprueba si se ha producido cualquier interrupción (suele ser una comprobación hardware), hecho indicado por la presencia de una señal de interrupción en un registro del procesador y enviada por la controladora de un dispositivo de E/S mediante un bus del sistema (en caso de ser una interrupción hardware de E/S). Si está pendiente una interrupción, el procesador suspende la ejecución del proceso actual y ejecuta una rutina ISR que realiza las acciones que se requieran, volviéndose después a reanudar la ejecución del proceso de usuario en el punto de la interrupción. En el caso de que no haya interrupciones pendientes, el procesador continúa con la fase de búsqueda y lee la siguiente instrucción del proceso actual.

Para apreciar la ganancia en eficiencia, considere la Figura 1.8, que es un diagrama de tiempo basado en el flujo de control de las figuras a) y b) del ejemplo de la impresora.

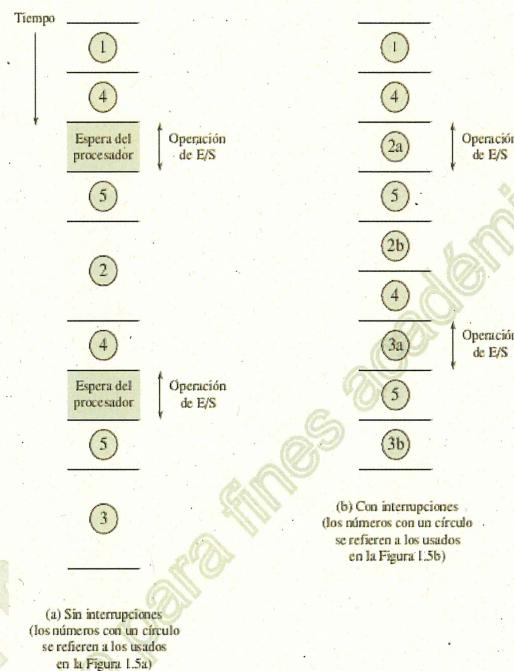


Figura 1.8. Temporización del programa: espera breve de E/S.

## 5.2 Interrupciones múltiples

El estudio realizado hasta el momento ha tratado solamente el caso de que se produzca una única interrupción. Supóngase, sin embargo, que se producen múltiples interrupciones, es decir, mientras se está tratando por *ISR* una interrupción se produce otra. Por ejemplo, un programa puede estar recibiendo datos de una línea de comunicación (Ethernet, Wifi) e imprimiendo resultados al mismo tiempo. La impresora generará una interrupción cada vez que completa una operación de impresión. El controlador de la línea de comunicación generará una interrupción cada vez que llega una unidad de datos. La unidad podría consistir en un único carácter o en un bloque, dependiendo de la naturaleza del protocolo de comunicaciones. En cualquier caso, es posible que se produzca una

interrupción de comunicación mientras se está procesando una interrupción de la impresora.

Se pueden considerar dos alternativas a la hora de tratar con múltiples interrupciones:

1) La primera es inhabilitar las interrupciones mientras que se está procesando una interrupción. Una interrupción inhabilitada significa simplemente que el procesador ignorará cualquier nueva señal de petición de interrupción. Si se produce una interrupción durante este tiempo, generalmente permanecerá pendiente de ser procesada, de manera que el procesador sólo la comprobará después de que se reabiliten las interrupciones. Por tanto, cuando se ejecuta un programa de usuario y se produce una interrupción, se inhabilitan las interrupciones inmediatamente. Después de que se completa la rutina de manejo de la interrupción *ISR*, se reabilitan las interrupciones antes de reanudar el programa de usuario, y el procesador comprueba si se han producido interrupciones adicionales. Esta estrategia es válida y sencilla, puesto que las interrupciones se manejan en estricto orden secuencial (Figura 1.12a).

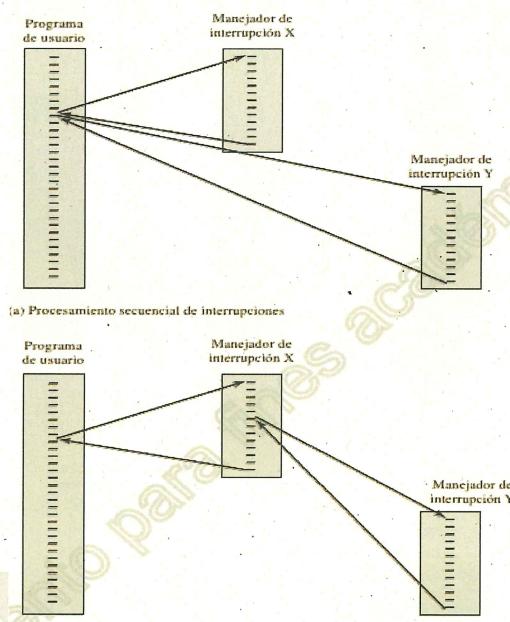


Figura 1.12. Transferencia de control con múltiples interrupciones.

La desventaja de la estrategia anterior es que no tiene en cuenta la prioridad relativa o el grado de urgencia de las interrupciones. Por ejemplo, cuando llegan datos por la línea de comunicación, se puede necesitar que se procesen rápidamente de manera que se deje sitio en los *buffers* de la controladora para otros datos que pueden llegar. Si el primer lote de datos no se ha procesado antes de que llegue el segundo (estamos ante un proceso prioritario), los datos pueden perderse porque el *buffer* de la controladora puede llenarse y desbordarse.

2) Una segunda estrategia es definir prioridades para las interrupciones y permitir que una interrupción de más prioridad cause que se interrumpa la ejecución de un manejador de una interrupción de menor prioridad (Figura 1.12b). Como ejemplo de esta segunda estrategia, considere un sistema con tres dispositivos de E/S: una impresora, un disco y una línea de

comunicación, con prioridades crecientes de 2, 4 y 5, respectivamente (suponga que un 5 significa más prioridad que un 2 y un 4).

La Figura 1.13, muestra una posible secuencia:

1. Un programa de usuario comienza en  $t = 0$ .
2. En  $t = 10$ , se produce una interrupción de impresora; se almacena la información de usuario y la ejecución continúa en la rutina de servicio de interrupción ISR de la impresora.
3. Mientras todavía se está ejecutando esta rutina, en  $t = 15$  se produce una interrupción del equipo de comunicaciones. Debido a que la línea de comunicación tiene una prioridad superior a la de la impresora, se sirve la petición de interrupción. Se interrumpe la ISR de la impresora, se almacena su estado, y la ejecución continúa con la ISR del equipo de comunicaciones.
4. Mientras se está ejecutando esta rutina, se produce una interrupción del disco ( $t = 20$ ). Dado que esta interrupción es de menor prioridad, simplemente se queda en espera, y la ISR de la línea de comunicación se ejecuta hasta su conclusión.
5. Cuando se completa la ISR de la línea de comunicación ( $t = 25$ ), se restituye el estado previo del proceso, que corresponde con la ejecución de la ISR de la impresora. Sin embargo, antes incluso de que pueda ejecutarse una sola instrucción de esta rutina, el procesador atiende la interrupción de disco de mayor prioridad y transfiere el control a la ISR del disco.
6. Sólo cuando se completa la rutina de disco ( $t = 35$ ), se reanuda la ISR de la impresora.
7. Cuando esta última rutina se completa ( $t = 40$ ), se vuelve finalmente el control al programa de usuario (previa restauración de su contexto).

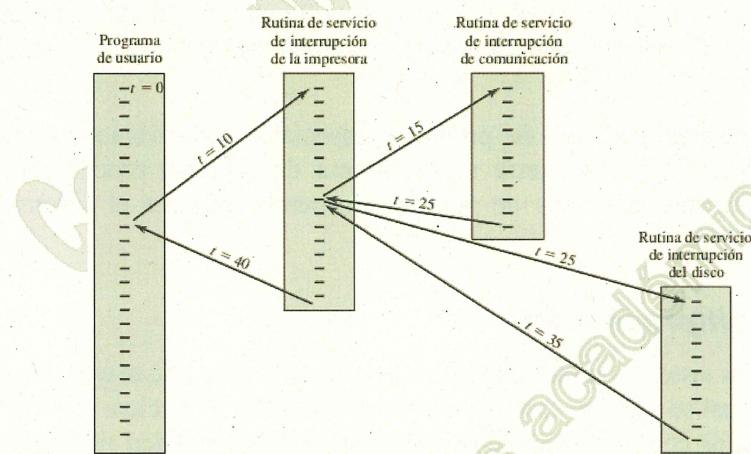


Figura 1.13. Ejemplo de secuencia de tiempo con múltiples interrupciones.

Puede pensar que el sistema puede dejar en inanición a otras interrupciones si por ejemplo continuamente se producen interrupciones con un alto nivel de prioridad. Para controlar eso está el planificador de dispositivos de E/S, de forma que no se sirva de manera infinita a interrupciones de

prioridad alta. Se estudiará lo que es la planificación en un tema dedicado a ello.

## **6 Arquitectura de un sistema informático desde el punto de vista del procesador**

De acuerdo con el número de procesadores de propósito general utilizados, un sistema informático se puede organizar de varias maneras diferentes, de las que se hablará brevemente a continuación.

### **6.1 Sistemas de un solo procesador**

Hasta hace relativamente poco tiempo, la mayor parte de los sistemas sólo usaban un procesador con un solo núcleo. En un sistema de un único procesador, hay una CPU principal capaz de ejecutar un conjunto de instrucciones de propósito general, incluyendo instrucciones de los programas de usuario. Además de esa CPU principal de propósito general, casi todos los sistemas disponen también de otros procesadores de propósito especial, los cuales pueden venir en forma de procesadores específicos de un dispositivo, como por ejemplo un disco, un teclado o una controladora gráfica. Estos procesadores específicos quitan carga al procesador principal realizando tareas concretas por él, como por ejemplo, transmitir datos entre la memoria principal y un dispositivo de E/S mientras que el procesador principal realiza otras tareas.

Todos estos procesadores de propósito especial ejecutan un conjunto limitado de instrucciones y no ejecutan programas de usuario (esto último solo lo hace el procesador principal). En ocasiones, el sistema operativo los gestiona, en el sentido de que les envía información sobre su siguiente tarea y monitoriza su estado. Por ejemplo, un microprocesador incluido en una controladora de disco recibe una secuencia de solicitudes procedentes de la CPU principal e implementa su propia cola de disco y su algoritmo de programación de tareas (se estudiarán algoritmos de planificación en un tema dedicado a ello). Este método libera a la CPU principal del trabajo adicional de planificar las tareas de disco.

La presencia de microprocesadores de propósito especial resulta bastante común (incluso en sistemas multiprocesador) y no convierte a un sistema de un solo procesador en un sistema multiprocesador. Si sólo hay una CPU de propósito general, entonces el sistema es de un solo procesador.

### **6.2 Sistemas multiprocesador**

La importancia de los sistemas multiprocesador está siendo cada vez mayor, hasta tal punto que casi ningún sistema actual se concibe con un solo procesador con un núcleo, sino que como mínimo posee un solo procesador con varios núcleos en su interior, que pueden realizar tareas de forma paralela.

Tales sistemas disponen de dos o más procesadores que se comunican entre sí compartiendo el bus de la computadora, la memoria y los dispositivos periféricos. Una CPU de doble núcleo, cuádruple núcleo, etc, tiene más de un procesador (también llamado núcleo) en su chip.

La ingeniería multinúcleo fue impulsada debido a los problemas provocados por la cantidad de

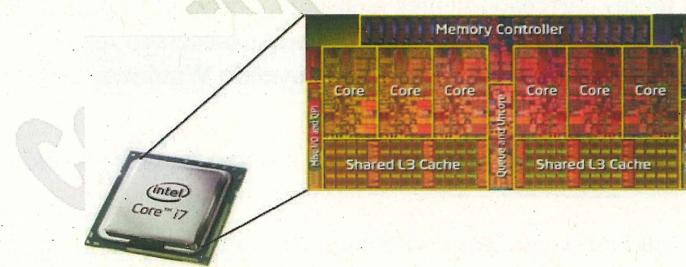
nanotransistores y su colocación cada vez más cercana entre ellos en un solo chip de CPU (la física cuántica y los electrones ya están en camino...). Aunque un chip con millones de nanotransistores ubicados bastante próximos entre si ayudaron a incrementar drásticamente el rendimiento del sistema, la estrecha proximidad de estos nanotransistores también aumentó la fuga de corriente y la cantidad de calor generada por el chip. Una solución consistía en crear un chip único, pero esta vez con dos o más núcleos. En otras palabras, sustituyeron un gran procesador único por dos núcleos de la mitad de tamaño, o por cuatro núcleos, cada uno con una cuarta parte del tamaño del procesador grande. Este diseño permitió que el chip del mismo tamaño produjera menos calor y ofreciera la oportunidad de permitir la realización de múltiples cálculos al mismo tiempo, sin embargo cada uno de los núcleo es más lento que un chip con un solo núcleo más grande.

Hay que decir que esta mejora tecnológica tiene un problema, y es que el sistema operativo debe tener mecanismos que permitan trabajar con varios núcleos o varios procesadores para que comparten RAM, periféricos etc, lo cual complica enormemente su implementación y diseño.

En la siguiente figura se presenta una placa base con 4 *slots* para alojar un procesador por *slot*, usados principalmente para tratar grandes volúmenes de datos con mucho procesamiento:



En la siguiente figura se presenta un sólo procesador con múltiples núcleos. Son los utilizados actualmente en sistemas personales:



Los sistemas multiprocesador presentan ventajas fundamentales:

- 1. Mayor rendimiento.** Al aumentar el número de procesadores, es de esperar que se realice más trabajo en menos tiempo, y así es, con lo cual ya es una gran ventaja. Sin embargo, la mejora en velocidad con  $N$  procesadores es *menor que  $N$* . Cuando múltiples procesadores cooperan en una tarea, cierta carga de trabajo se emplea en conseguir que todas las partes funcionen correctamente. Esta carga de trabajo, más la competición por los recursos compartidos, reducen la ganancia esperada por añadir procesadores adicionales.

2. **Economía de escala.** Los sistemas multiprocesador pueden resultar más baratos que su equivalente con múltiples sistemas de un solo procesador, ya que pueden compartir periféricos, almacenamiento masivo y fuentes de alimentación. Si varios programas operan sobre el mismo conjunto de datos, es más barato almacenar dichos datos en un disco y que todos los procesadores los compartan, que tener muchas computadoras con discos locales y muchas copias de los datos.

Los sistemas multiprocesador utilizados son de dos tipos (simétrico y asimétrico). Algunos sistemas usan el **multiprocesamiento asimétrico**, en el que cada procesador se asigna a una tarea específica. Un procesador maestro controla el sistema y el resto de los procesadores esperan que el maestro les dé instrucciones o tienen asignadas tareas predefinidas. Este esquema define una relación maestro-esclavo. El procesador maestro planifica el trabajo de los procesadores esclavos y se lo asigna, desempeña actividades de administración de almacenamiento y ejecuta todos los programas de control.

La ventaja más importante de esta configuración es su simpleza, sin embargo presenta tres desventajas serias:

- Su confiabilidad no es mayor que la de un sistema con procesador único, porque si falla el procesador maestro falla todo el sistema.
- Puede llevar a un uso deficiente de los recursos, porque si un procesador esclavo se libera mientras el procesador maestro está ocupado, el esclavo tiene que esperar hasta que el maestro esté libre y le pueda asignar trabajo.
- Incrementa el número de interrupciones, porque todos los procesadores esclavos deben interrumpir al procesador maestro cada vez que requieren la intervención del sistema operativo, como por ejemplo para solicitudes de E/S. Esto crea largas colas a nivel del procesador maestro que pueden dar lugar a un cuello de botella.

Los sistemas más comunes utilizan el **multiprocesamiento simétrico (SMP)**, en el que cada procesador realiza todas las tareas correspondientes al sistema operativo. En un sistema SMP, todos los procesadores son iguales; no existe una relación maestro-esclavo entre los procesadores. Prácticamente todos los sistemas operativos modernos, incluyendo Windows, Mac OS y Gnu/Linux, proporcionan soporte para SMP.

Se puede definir un multiprocesador simétrico como un sistema de computación aislado con las siguientes características:

- Tiene múltiples procesadores o núcleos contenidos dentro del mismo procesador.
- Estos procesadores comparten las mismas utilidades de memoria principal y de E/S, interconectadas por un bus de comunicación u otro esquema de conexión interna.
- Todos los procesadores pueden realizar las mismas funciones, de ahí el término simétrico.
- La existencia de múltiples procesadores es transparente al usuario. El sistema operativo se encarga de planificar los hilos o procesos en procesadores individuales, y de la sincronización entre los procesadores.

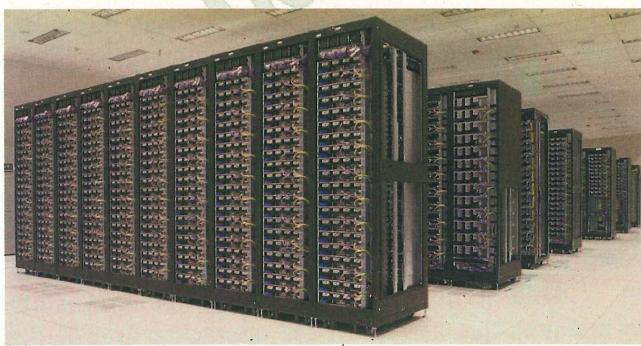
SMP tiene una ventaja fundamental sobre las arquitecturas monoprocesador (con un solo núcleo), además de las ventajas de **Mayor rendimiento** y **Economía de escala** comunes entre asimétrico y simétrico:

- Es más **confiable**, ya que si un procesador deja de funcionar no se cae el sistema entero. El sistema operativo debe tener mecanismos que permitan adecuar la carga de trabajo a la nueva situación si uno cae, pero el sistema puede continuar funcionando con un rendimiento reducido.
- Aumento de la **disponibilidad**. La definición de disponibilidad tiene que ver con la capacidad de un servicio o de un sistema, a ser accesible y utilizable por los usuarios o programas autorizados, cuando estos lo requieran. Por ejemplo, los sistemas asimétricos pueden que son tengan disponibilidad en un momento si el procesador maestro está realizando muchas tareas y no puede atender a una nueva petición de un usuario para delegarla a un procesador esclavo. Este caso concreto provocaría un cuello de botella en el procesador maestro. En los procesadores simétricos esto no ocurre, a no ser que todos los procesadores estén altamente ocupados.

Es importante volver a recalcar que estas características son beneficios potenciales, no garantizados. El sistema operativo debe proporcionar herramientas y funciones para explotar el paralelismo en un sistema SMP, lo que hace más complejo al sistema operativo. Esto es una desventaja que da lugar a dos desafíos generales: Cómo conectar los procesadores y cómo orquestar (planificar) su interacción.

### 6.3 Sistemas en cluster

Otro tipo de sistema con múltiples CPU es el sistema en *cluster*. Como los sistemas multiprocesador, los sistemas en *cluster* utilizan múltiples CPU para llevar a cabo el trabajo, y cada CPU puede tener varios núcleos. Los sistemas en *cluster* se diferencian de los sistemas de multiprocesamiento en que están formados por dos o más sistemas individuales acoplados. La definición del término *cluster* generalmente aceptada es aquella en que las computadoras en *cluster* comparten el almacenamiento y se conectan entre sí a través de una red de área local (LAN, *local area network*), de tal forma que el conjunto es visto como un único ordenador, más potente que los comunes de escritorio.



Los *clusters* son usualmente empleados para mejorar el rendimiento y/o la disponibilidad por encima de la que es provista por un solo computador.

La construcción de ordenadores tipo *cluster* es relativamente económica debido a su flexibilidad: pueden tener todos la misma configuración de hardware y sistema operativo (*cluster* homogéneo), diferente rendimiento pero con arquitecturas y sistemas operativos similares (*cluster* semihomogéneo), o tener diferente hardware y sistema operativo (*cluster* heterogéneo). Pero para que un *cluster* funcione como tal, no basta solo con conectar entre sí los ordenadores, sino que es necesario proveer un sistema software de manejo del *cluster*, el cual se encargue de interactuar con el usuario y los procesos que corren en él para optimizar el funcionamiento.

El término *cluster* tiene diferentes connotaciones:

- **Clusters de alto rendimiento:** Son *clusters* en los cuales se ejecutan tareas que requieren de gran capacidad computacional, grandes cantidades de memoria, o ambos a la vez. El llevar a cabo estas tareas puede comprometer los recursos del *cluster* por largos períodos de tiempo.
- **Clusters de alta disponibilidad:** Son *clusters* cuyo objetivo de diseño es el de proveer disponibilidad y confiabilidad. Estos *clusters* tratan de brindar la máxima disponibilidad de los servicios que ofrecen. La confiabilidad se provee mediante software que detecta fallos y permite recuperarse frente a los mismos, mientras que en hardware se evita tener un único punto de fallo.
- **Clusters de alta eficiencia:** Son *clusters* cuyo objetivo de diseño es el ejecutar la mayor cantidad de tareas en el menor tiempo posible. Existe independencia de datos entre las tareas individuales, es decir, no se necesita que una tarea termine para poder comenzar otra. El retardo entre los nodos del *cluster* no se considera un gran problema.

## 7 Multiprogramación

Uno de los aspectos más importantes de los sistemas operativos es la capacidad para multiprogramar. La **multiprogramación** incrementa el uso de la CPU de manera que ésta no quede ociosa de trabajos.

La idea es la siguiente:

El sistema operativo mantiene en memoria principal, de forma simultánea, varios trabajos, y empieza a ejecutar uno de ellos. Eventualmente, el trabajo puede tener que esperar a que se complete alguna otra tarea, como por ejemplo una operación de E/S (que el usuario deba introducir una clave por teclado), o se queda a la espera de que se produzca un evento concreto (espera a que la CPU alcance cierta temperatura, a que la tasa de bajada de red sea una cantidad determinada, a que un periférico se quede libre, que el disco duro le devuelva información sobre un fichero que se requiere, etc). Pues bien, si el sistema no fuera multiprogramado, la CPU se quedaría inactiva hasta que una operación de E/S concluyese o el evento esperado se produjese y el trabajo en ejecución pudiera continuar. Por el contrario, en un sistema multiprogramado, el sistema operativo simplemente cambia de trabajo y ejecuta otro mientras que se realiza esa operación de E/S o se espera a ese evento concreto. Siempre que la CPU tiene que esperar se conmuta a otro trabajo que esté listo para ejecución, y así sucesivamente. A qué trabajo se cambia es decisión del planificador

que el sistema utilice (se estudiará en uno de los temas de la asignatura). De esta forma, mientras haya al menos un trabajo que necesite ejecutarse, además del trabajo actual en ejecución, la CPU nunca estará inactiva y ociosa.

La figura 2.5 muestra un ejemplo de multiprogramación en un sistema con un solo procesador, pero simulando que en memoria principal hay 1, 2 o 3 procesos listos para ejecución. Se entiende en este ejemplo que hay un solo procesador con un solo núcleo. Observe como se va reduciendo en cada ejemplo el tiempo ocioso del procesador.

En la literatura también puede encontrar la palabra **multitarea** refiriéndose a la multiprogramación, pero entendiéndose multitarea como la capacidad de un sistema de permitir que varios procesos se ejecuten al mismo tiempo. En realidad, la multitarea solo se puede producir de manera real en un sistema con dos procesadores, en un sistema con un solo procesador, lo que se estaría produciendo mientras usted escucha música y a la vez escribe una carta en un procesador de textos, es conmutar muy rápidamente entre procesos, fraccionando el tiempo de uso entre ellos.

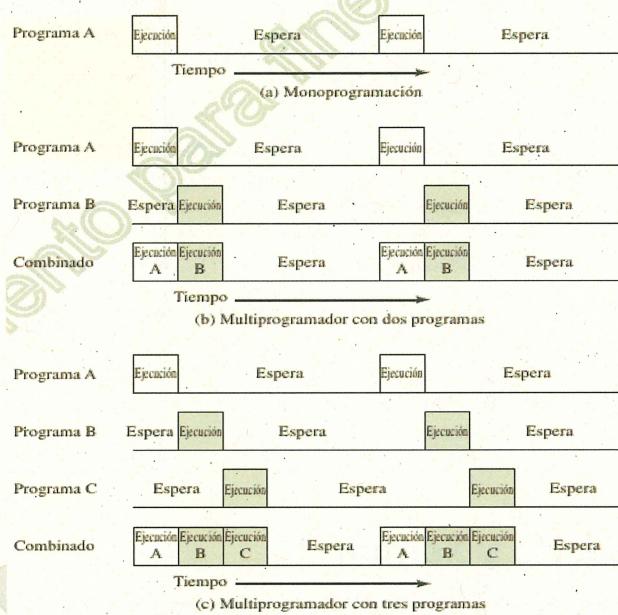


Figura 2.5. Ejemplo de multiprogramación.

## 8 Multiprocesamiento

En un entorno multiprocesador, ni que decir tiene que también se utiliza la multiprogramación, de forma que cada procesador pueda beneficiarse de ello y no quede inactivo. Lo que no debemos confundir, es el **multiprocesamiento o multiproceso**, el cual se da en sistemas con mas de un procesador. Se puede definir de manera formal como el uso o ejecución de múltiples programas concurrentes (en el mismo tiempo, a la misma vez) en un sistema, en lugar de un único proceso en un instante determinado (monoprocesamiento o monoproceso).

Resumamos entonces: Con la multiprogramación, sólo un proceso puede ejecutar a la vez; mientras tanto, el resto de los procesos esperan por el procesador. Con multiproceso, más de un

proceso puede ejecutarse simultáneamente, cada uno de ellos en un procesador diferente. Pero a pesar del trabalenguas recuerde, ¡en un sistema con multiprocesamiento también hay multiprogramación!

En la Figura 2.12a se puede observar la multiprogramación en un sistema con un solo procesador (sobreentendemos que con un solo núcleo), mientras que en la Figura 2.12b se observa multiprocesamiento en un sistema con dos procesadores. En este último caso también podría haber multiprogramación si el Proceso 3 estuviera listo para ejecutar desde el instante 0.

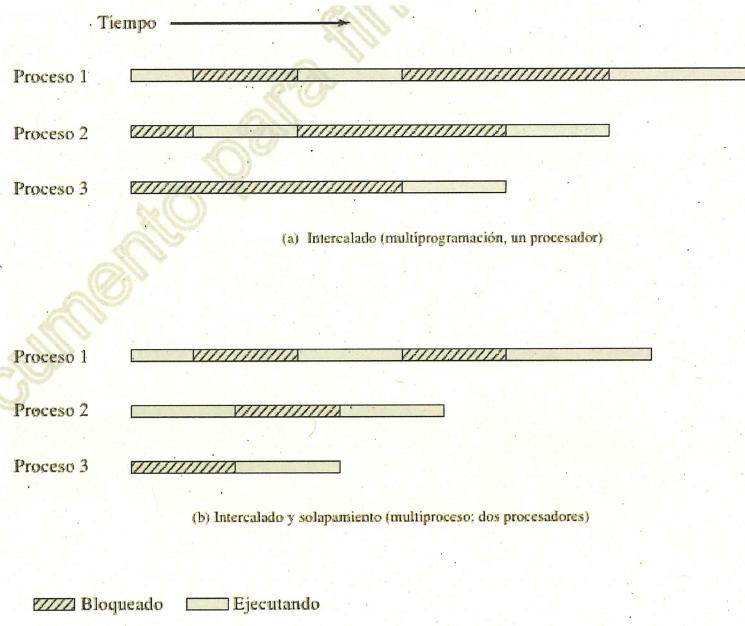


Figura 2.12. Multiprogramación y multiproceso.

## 9 Modo dual (usuario y kernel)

Un sistema operativo siempre está esperando a que ocurran sucesos. Los sucesos se indican mediante la ocurrencia de una interrupción o una excepción. Una excepción es una interrupción generada por software, debida a un error, por ejemplo, una división por cero o un acceso a memoria no válido. Para cada tipo de interrupción, diferentes segmentos de código del sistema operativo determinan qué acción hay que llevar a cabo.

El código del propio sistema operativo, es decir, su conjunto de instrucciones para gestionar recursos y proporcionar servicios, se encuentra codificado en lenguaje máquina. Dado que el sistema operativo y los procesos y usuarios comparten los recursos hardware y software del sistema informático, necesitamos asegurar que un error que se produzca en un programa de usuario sólo genere problemas en el programa que se estuviera ejecutando, y no en el control del hardware o incluso en otros usuarios y programas.

Imaginemos que el usuario, a través de un programa de aplicación, pudiera hacer cambios en tiempo real de las rutinas o instrucciones máquina que rigen el funcionamiento del sistema operativo y que interactúan con el hardware, imaginemos que pudiera cambiar el manejador de interrupciones, la forma en que se produce comunicación con un dispositivo de entrada-salida; imaginemos que pudiera cambiar la codificación del sistema operativo o acceder a ella para utilizarla a su manera y gusto. Si esto sucediera, se podría hacer un uso indebido del sistema, en última instancia del hardware, produciendo comportamientos inesperados y pudiendo corromper el funcionamiento y gestión de los recursos, programas y usuarios.

Por tanto, para que el sistema informático funcione con seguridad y adecuadamente, hay que impedir que los programas de usuario puedan realizar libremente ciertas operaciones que puedan llevar a conflictos y a un mal uso del sistema, como por ejemplo, invocaciones u operaciones que tengan que ver con:

- Utilización de la CPU por parte de un proceso todo el tiempo que lo requiera, dejando al resto de procesos y usuarios en inanición.
- Acceso a zonas de memoria restringidas.
- Acceso directo a los dispositivos de E/S.
- Uso inadecuado de las interrupciones del sistema.

Para ello, los diseñadores han buscado una solución basada en un modo dual de operación. Eso hace que haya que distinguir entre los modos de ejecución del procesador:

- El **modo núcleo, kernel, supervisor o privilegiado**, asociado al procesamiento de rutinas e instrucciones del sistema operativo.
- El **modo usuario**, asociado al procesamiento de rutinas e instrucciones de los programas de usuario.

De esta manera, determinadas instrucciones o rutinas del sistema operativo solo se pueden ejecutar en modo núcleo. El núcleo es la parte del sistema operativo que engloba las funciones y servicios más importantes del sistema y que deben protegerse. Éstas instrucciones y rutinas incluirían aquellas que tienen que ver con el manejo de la E/S, la CPU y la memoria del sistema. Es una parte relativamente pequeña del sistema, pero la más utilizada, por ello suele residir en memoria principal de forma continua, mientras que el resto del sistema operativo se carga cuando es necesario. El núcleo del sistema se carga siempre en una zona de memoria de acceso restringido a programas de usuario.

La Tabla 3.7 lista las funciones básicas que normalmente se encuentran dentro del núcleo del sistema operativo:

Tabla 3.7. Funciones típicas de un núcleo de sistema operativo.

| Gestión de procesos  |
|--|
| <ul style="list-style-type: none"> <li>• Creación y terminación de procesos</li> <li>• Planificación y activación de procesos</li> <li>• Intercambio de procesos</li> <li>• Sincronización de procesos y soporte para comunicación entre procesos</li> <li>• Gestión de los bloques de control de proceso</li> </ul> |
| Gestión memoria  |
| <ul style="list-style-type: none"> <li>• Reserva de espacio direcciones para los procesos</li> <li>• Swapping</li> <li>• Gestión de páginas y segmentos</li> </ul>   |
| Gestión E/S  |
| <ul style="list-style-type: none"> <li>• Gestión de buffers</li> <li>• Reserva de canales de E/S y de dispositivos para los procesos</li> </ul>  |
| Funciones de soporte   |
| <ul style="list-style-type: none"> <li>• Gestión de interrupciones</li> <li>• Auditoria</li> <li>• Monitorización</li> </ul>   |

Dicho esto, aparecen dos cuestiones: ¿cómo conoce el procesador en qué modo está ejecutando? Y, ¿cuándo se cambia de modo? En lo referente la primera cuestión, existe típicamente un bit en la palabra de estado de programa (PSW) que indica el modo de ejecución. Este bit se cambia como respuesta a determinados eventos. Con respecto a la segunda cuestión, cuando ocurre una interrupción, una excepción o una llamada al sistema, el modo de ejecución se cambia a modo núcleo y; tras la finalización del servicio, el modo se fija de nuevo a modo usuario.

De esta manera el hardware detecta los errores de violación de los modos y es, normalmente, el sistema operativo el que se encarga de tratarlos. Si un programa de usuario falla de alguna forma, como por ejemplo haciendo un intento de ejecutar una instrucción ilegal privilegiada o de acceder a una zona de memoria que no esté en el espacio de memoria del usuario, entonces el hardware envía una excepción al sistema operativo. La excepción transfiere el control al sistema operativo a través del vector de interrupción, igual que cuando se produce una interrupción. Entonces el sistema operativo proporciona un mensaje de error apropiado, el cual puede volcarse en memoria. Normalmente, el volcado de memoria se escribe en un archivo con el fin de que el usuario o programador puedan examinarlo y quizás corregir y reiniciar el programa.

La Figura 1.8 muestra gráficamente un cambio de modo. Como dato curioso, en cualquier S.O. moderno, como lo es Gnu/Linux, el procesador alterna entre ambos modos al menos unas cuantas miles de veces por segundo.

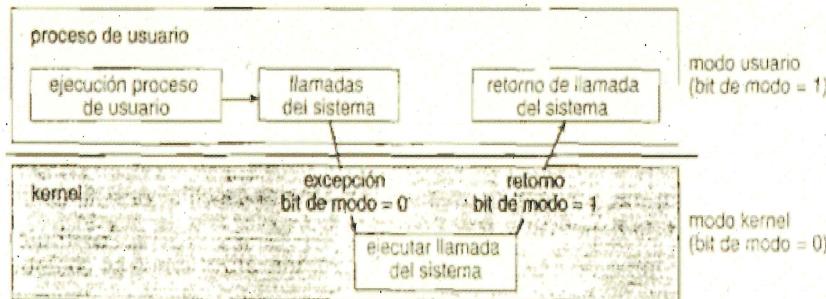


Figura 1.8 Transición de modo usuario a modo kernel.

La falta de un modo dual soportado por hardware puede dar lugar a serios defectos en un sistema operativo. Por ejemplo, MS-DOS, que fue escrito para la arquitectura 8088 de Intel no disponía de bit de modo y, por tanto, tampoco del modo dual. Un programa de usuario malintencionado podía corromper el sistema operativo sobreescritiendo sus archivos, y múltiples programas podrían escribir en un dispositivo al mismo tiempo, con resultados posiblemente desastrosos.

## 10 Llamadas al sistema

Las llamadas al sistema proporcionan un interfaz entre un programa y el sistema operativo para poder invocar los servicios que éste ofrece, ya que como hemos comentado antes con los modos duales, el usuario, por protección del sistema, no puede acceder o no tiene privilegios directos sobre los recursos que gestiona el sistema operativo.

Las llamadas al sistema pueden agruparse de forma muy general en cinco categorías principales control de procesos, manipulación de archivos, manipulación de dispositivos, mantenimiento de información y comunicaciones. Estas llamadas, generalmente, están disponibles para el programador como rutinas escritas en C y C++, aunque determinadas tareas de bajo nivel, como por ejemplo aquéllas en las que se tiene que acceder directamente al hardware, pueden necesitar escribirse con instrucciones de lenguaje ensamblador. La ejecución de las llamadas en sí, se realizan en modo núcleo, ejecutándose instrucciones máquina que se encuentran en la memoria principal, concretamente en una zona de memoria restringida que se usa solo para alojar al núcleo del sistema operativo.

Antes de ver cómo pone un sistema operativo a nuestra disposición las llamadas al sistema, vamos a ver un ejemplo para ilustrar cómo se usan esas llamadas. Suponga que deseamos escribir un programa sencillo para leer datos de un archivo y copiarlos en otro archivo:

El primer dato de entrada que el programa va a necesitar son los nombres de los dos archivos, el de entrada y el de salida. Estos nombres pueden especificarse de muchas maneras, dependiendo del diseño del sistema operativo; un método consiste en que el programa pida al usuario que introduzca los nombres de los dos archivos. Este método, requerirá por tanto, una secuencia de llamadas al sistema:

Primero hay que mostrar un mensaje en el *shell* de comandos (llamada al sistema) y luego leer del teclado (llamada al sistema) los caracteres que especifican los dos archivos.

Una vez que se han obtenido los nombres de los dos archivos, el programa debe abrir el archivo de entrada (llamada al sistema) y crear el archivo de salida (llamada al sistema).

Cuando el programa intenta abrir el archivo de entrada, puede encontrarse con que no existe ningún archivo con ese nombre, o que está protegido contra accesos. En estos casos, el programa debe escribir un mensaje en la consola (llamada al sistema) y terminar de forma anormal (llamada al sistema). Si el archivo de entrada existe, entonces se debe crear un nuevo archivo de salida. En este caso, podemos encontrarnos con que ya existe un archivo de salida con el mismo nombre; esta situación puede hacer que el programa termine (llamada al sistema) o podemos borrar el archivo existente (llamada al sistema) y crear otro (llamada al sistema).

Una vez que ambos archivos están definidos, hay que ejecutar un bucle que lea del archivo de entrada (llamada al sistema) y escriba en el archivo de salida (llamada al sistema). Cada lectura y

escritura debe devolver información de estado relativa a las distintas condiciones posibles de error. En la lectura, el programa puede encontrarse con que ha llegado al final del archivo o con que se ha producido un fallo de hardware. En la operación de escritura pueden producirse varios errores, dependiendo del dispositivo de salida (espacio de disco insuficiente, la impresora no tiene papel, etc.)

Finalmente, después de que se ha copiado el archivo completo, el programa cierra ambos archivos (llamada al sistema), escribe un mensaje en la consola (llamada al sistema) y, por último, termina normalmente (llamada al sistema).

Lo que este ejemplo quiere hacerle ver es que incluso los programas más sencillos hacen un uso intensivo del sistema operativo y de servicios y rutinas implementadas en su núcleo. Normalmente, los sistemas ejecutan miles de llamadas al sistema por segundo, y son transparentes al programador y al usuario final.

Una secuencia abstracta de llamadas al sistema, relativas al ejemplo redactado, se muestra en la Figura 2.1.

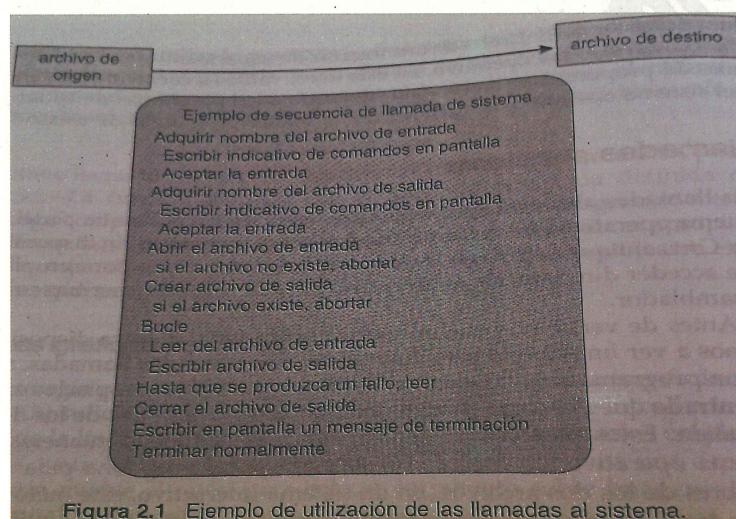
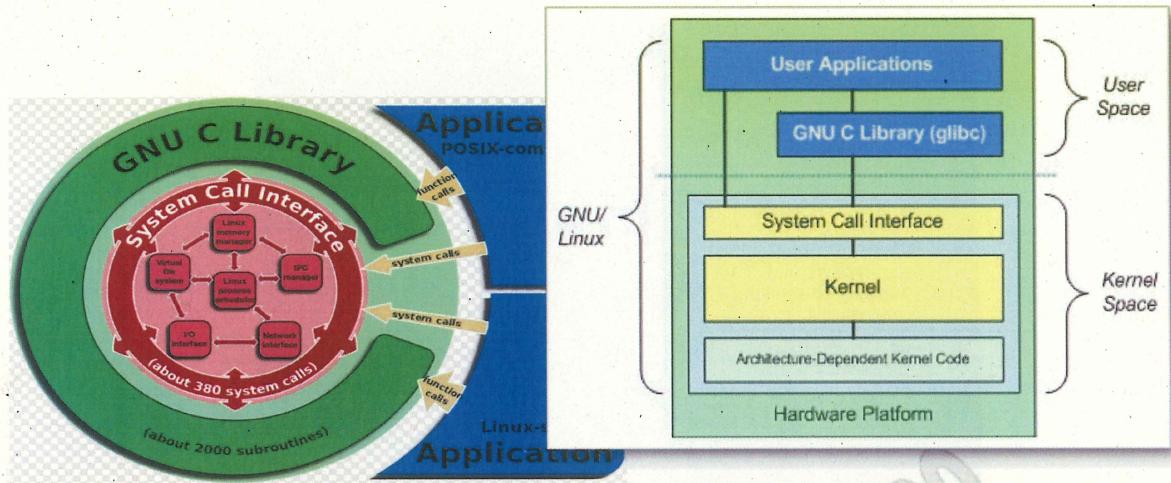


Figura 2.1 Ejemplo de utilización de las llamadas al sistema.

Sin embargo, la mayoría de los programadores no ven este nivel de detalle. Normalmente, los desarrolladores de aplicaciones diseñan sus programas utilizando una API (*application programming interface*, interfaz de programación de aplicaciones). La API especifica un conjunto de funciones que el programador de aplicaciones puede usar, indicándose los parámetros que hay que pasar a cada función y los valores de retorno que el programador debe esperar. Dos de las API más usuales disponibles para programadores de aplicaciones son la API Win32 para sistemas Windows y la API de la biblioteca estándar de C, *glibc*, para sistemas basados en POSIX (prácticamente todas las versiones de UNIX, Gnu/Linux y Mac OS).



Las funciones que conforman una API de alto nivel invocan a las llamadas al sistema por cuenta del programador de la aplicación. Por ejemplo, la función *CreateProcess()* de Win32 (crea un nuevo proceso) lo que hace realmente, es invocar la llamada nativa al sistema *NTCreateProcess()* del *kernel* de Windows. Lo mismo pasa con Gnu/Linux, pero los nombres de las llamadas son diferentes.

¿Por qué un programador de aplicaciones preferiría usar una API en lugar de invocar las propias llamadas al sistema? Existen varias razones para que sea así:

- Una ventaja de programar usando una API está relacionada con la portabilidad. Un programador de aplicaciones diseña un programa usando una API cuando quiere poder compilar y ejecutar su programa en cualquier sistema que soporte la misma API.
- Resulta más difícil por parte de los programadores de aplicaciones trabajar con las propias llamadas al sistema, además de que exige un grado mayor de detalle con respecto a usar una API. De todos modos, existe una fuerte correlación entre invocar una función de la API y su llamada al sistema asociada disponible en el kernel. De hecho, muchas de las API Win32 y *glibc* son similares a las llamadas al sistema nativas proporcionadas por los sistemas operativos UNIX, Gnu/Linux y Windows.

Quien realiza una llamada al sistema no tiene por qué saber nada acerca de cómo se implementa dicha llamada o qué es lo que ocurre durante su ejecución. Tan solo necesita ajustarse a lo que la API especifica y entender lo que hará el sistema operativo como resultado de la ejecución de dicha llamada al sistema. Por tanto, la API oculta al programador la mayor parte de los detalles de la interfaz del sistema operativo. La Figura 2.3, que ilustra cómo gestiona el sistema operativo una aplicación de usuario invocando la llamada nativa al sistema *open()*. Esta llamada sigue el estándar POSIX, y es la que se usa en los sistemas que lo implementan.

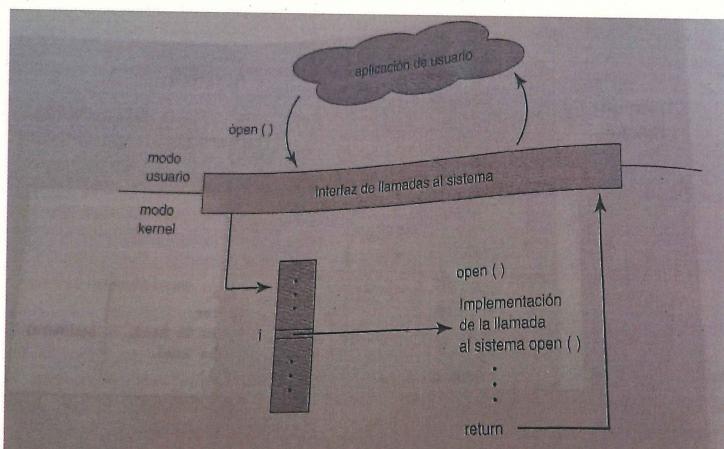


Figura 2.3 Gestión de la invocación de la llamada al sistema `open()` por parte de una aplicación.

En la Figura 2.6, se muestra otro ejemplo de una llamada en C para escribir por la salida estándar del sistema. Suponga un programa en C que invoque la sentencia `printf()`. La biblioteca de C intercepta esta llamada e invoca las llamadas nativas al núcleo que se necesiten, en este caso solo hace falta invocar a la llamada del sistema `write()`. Posteriormente, la biblioteca de C toma el valor que devuelve `write()` y lo pasa de nuevo al programa de usuario.

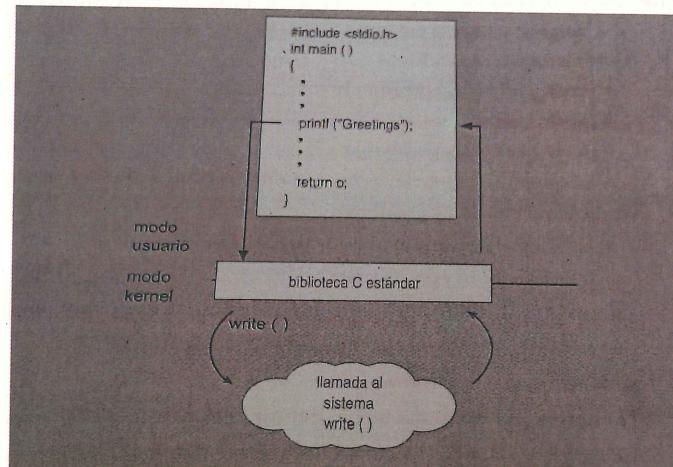


Figura 2.6 Tratamiento de la instrucción `write()` en la biblioteca C.

## 10.1 Trap para acceso al núcleo

Una vez presentado el concepto de llamada al sistema, a continuación se muestra a un nivel más bajo de abstracción, qué ocurre cuando se produce una invocación:

1. Cuando se produce una llamada nativa al sistema, los parámetros asociados a la misma se cargarían en la pila.

2. Posteriormente, se ejecuta una instrucción especial del sistema operativo que se llama *trap*, y que conlleva un cambio de modo, de modo usuario a modo núcleo. Al ejecutarse la instrucción *trap* se ejecuta un programa que examina la rutina nativa invocada y sus parámetros, y busca en una tabla o vector de rutinas si existe, además de la dirección del lugar del núcleo donde se encuentra para proceder a ejecutarla. Hay que decir, que cada llamada nativa al sistema tiene asociado un número que se mantiene en una tabla indexada en memoria principal.
3. Tras identificar que existe una rutina para la llamada realizada y que los parámetros son correctos, se procede a ejecutarla.
4. Posteriormente el programa *trap* solicita un código de estado almacenado en un registro, este registro señala si la llamada tuvo éxito o no y ejecuta una instrucción de tipo RETURN FROM TRAP a nivel de núcleo para regresar el control a la rutina de biblioteca a modo usuario. Se devuelve también un parámetro resultado en el caso de que se tenga que devolver algo, además del éxito o fracaso de la llamada al sistema.
5. Cuando finalizan esas acciones, la rutina de biblioteca a nivel de usuario descarga la pila y comprueba el resultado de la ejecución de la petición al sistema, se lo devuelve al usuario y se continua por la siguiente linea de código del programa.

## 11 Intérprete de comandos

Algunos sistemas operativos incluyen el intérprete de comandos en el *kernel*. En los sistemas que disponen de varios intérpretes de comandos entre los que elegir, los intérpretes se conocen como *shells*. Por ejemplo, en los sistemas UNIX y Gnu/Linux, hay disponibles varias *shells* diferentes entre las que un usuario puede elegir, incluyendo la *shell Bourne*, la *shell C*, la *shell Bourne-Again*, la *shell Korn*, etc. La mayoría de las *shells* proporcionan funcionalidades similares, existiendo sólo algunas diferencias menores, casi todos los usuarios seleccionan una *shell* u otra basándose en sus preferencias personales.

La función principal del intérprete de comandos es obtener y ejecutar el siguiente comando especificado por el usuario. Muchos de los comandos que se proporcionan en este nivel se utilizan para manipular archivos: creación, borrado, listado, impresión, copia, ejecución, etc. Estos comandos pueden implementarse de dos formas generales:

- Uno de los métodos consiste en que el propio intérprete de comandos contiene el código que el comando tiene que ejecutar. Por ejemplo, un comando para borrar un archivo puede hacer que el intérprete de comandos salte a una sección de su código que configura los parámetros necesarios y realiza las apropiadas llamadas al sistema. En este caso, el número de comandos que puede proporcionarse determina el tamaño del intérprete de comandos, dado que cada comando requiere su propio código de implementación.
- Un método alternativo, utilizado por los sistemas basados en UNIX y otros sistemas operativos, implementa la mayoría de los comandos a través de una serie de programas del sistema. En este caso, el intérprete de comandos no "entiende" el comando, sino que simplemente lo usa para identificar el archivo que hay que cargar en memoria y ejecutar.

Por tanto, el comando UNIX “*rm file.txt*” para borrar un archivo buscaría un archivo llamado *rm*, cargaría el archivo en memoria y lo ejecutaría, pasándole el parámetro *file.txt*. La función asociada con el comando *rm* queda definida completamente mediante el código contenido en el archivo *rm*. De esta forma, los programadores pueden añadir comandos al sistema fácilmente, creando nuevos archivos con los nombres apropiados. El programa intérprete de comandos, que puede ser pequeño, no tiene que modificarse en función de los nuevos comandos que se añaden, lo que hace es buscar ese comando en el directorio desde donde se hace una invocación y en la variable PATH del sistema.

## 12 Arranque del sistema

Para que una computadora comience a funcionar, por ejemplo cuando se enciende o se reinicia, es necesario que tenga un programa de inicio que ejecutar. Normalmente, se almacena en una memoria ROM (*read only memory*, memoria de sólo lectura) o en una memoria EEPROM (*electrically erasable programmable read only memory*, memoria de sólo lectura programable y eléctricamente borrable), y se conoce con el término general *firmware* o *Bios*. Una memoria tipo ROM resulta adecuada, ya que no necesita inicialización y no puede verse infectada por un virus informático. Si se usa una ROM, para cambiar el sistema de arranque hay que cambiar el *chip* en si, si embargo si la computadora posee una EEPROM puede actualizarse el *firmware*.

El propósito del *Bios* es identificar y diagnosticar los dispositivos del sistema, como la CPU, la RAM, las controladoras, la tarjeta de vídeo, el teclado, la unidad de disco duro, la unidad de disco óptico y otro hardware básico; de forma que el entorno quede preparado para poder cargar el núcleo o *kernel* del sistema operativo en la RAM.

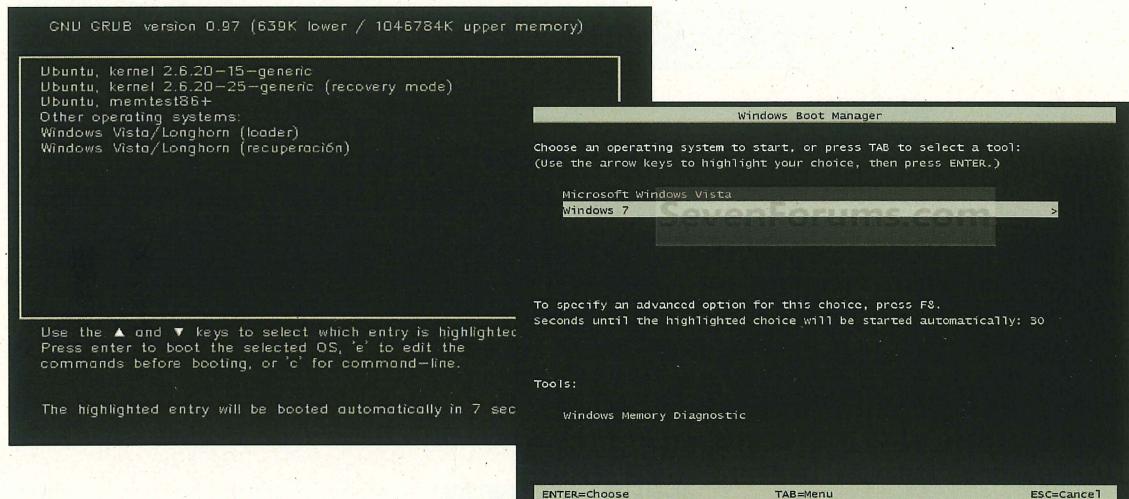
Si se pasan las pruebas de diagnóstico satisfactoriamente, el programa puede continuar con la secuencia de arranque. En este segundo paso el *Bios* hace que se comience a ejecutar un programa llamado **gestor de arranque o cargador de arranque de segunda etapa**. Los cargadores de arranque suelen encontrarse en el primer sector (bloque 0) del dispositivo de arranque (disco duro, CD-ROM, Pendrive). Para que comience a ejecutar el gestor de arranque, la *Bios* busca en ese sector y lo carga en memoria principal, de forma que se comienza a ejecutar el código que hay en dicho bloque de arranque.

El gestor de arranque puede ser lo suficientemente complejo como para cargar el sistema operativo completo en memoria e iniciar su ejecución. Normalmente, se trata de un código que cabe en un solo bloque de disco y que únicamente conoce la dirección del disco y la longitud del resto del programa de arranque. Un disco que tiene una partición marcada como de arranque en su sector cero, se denomina **disco de arranque o disco del sistema**.

Cuando el gestor de arranque empieza a ejecutarse comienza a explorar el sistema de archivos para localizar el *kernel* del sistema operativo y cargarlo en memoria principal. Una vez cargado el *Kernel*, el sistema operativo comienza el primer proceso, como por ejemplo “*init*” en sistemas GNU/Linux, y espera a que se produzca algún suceso. Sólo en esta situación se dice que el sistema está en ejecución.

Como ejemplos de cargadores de arranque tenemos *Lilo* y *Grub* en sistemas *UNIX*, o *NT Loader* y *Windows Boot Manager* en sistemas Windows. Al cargarse y ejecutarse ese software se le da el control del computador al sistema operativo, que comienza a ejecutarse. En las siguientes figuras se

muestran arranques duales gestionados por *Grub* y *Windows Boot Manager*. Si su sistema no posee más de un sistema operativo, lo normal es que estos cargadores ni siquiera aparezcan de manera visual en el arranque.



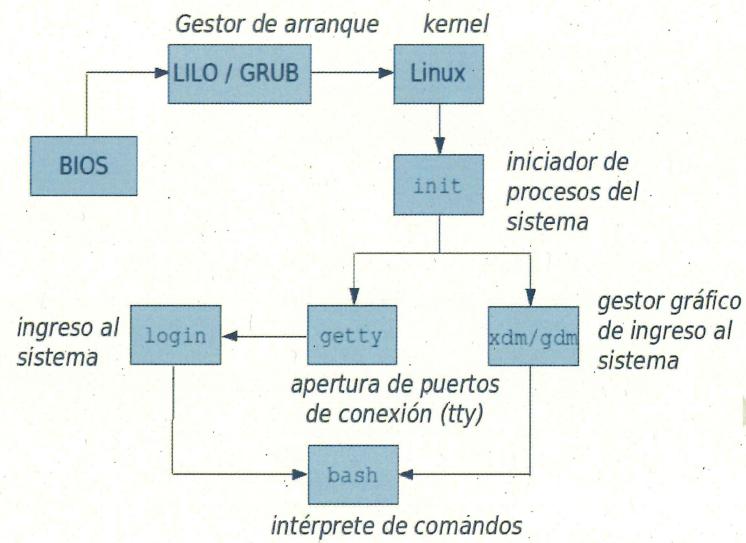
Un problema general con el *firmware* es que ejecutar código en él es más lento que ejecutarlo en RAM. Un último problema con el firmware es que es relativamente caro, por lo que normalmente sólo está disponible en pequeñas cantidades dentro de un sistema.

Continuando con el proceso *init* en los sistemas GNU/Linux, la lista exacta de tareas que realiza puede variar dependiendo de la distribución, pero aquí se muestran algunos ejemplos:

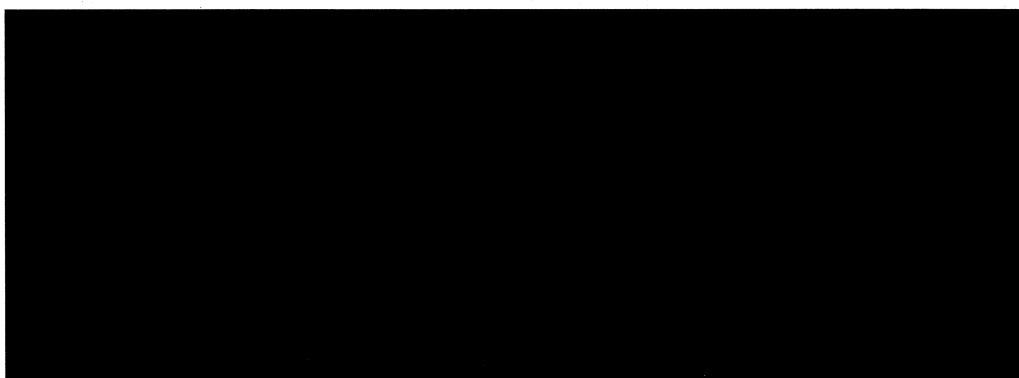
- Chequeo de los sistemas de archivos.
- Borrado del contenido de /tmp.
- Inicialización de los demonios y servicios esenciales para el correcto funcionamiento del sistema.
- Inicialización de procesos a petición del usuario, ya sea mediante GUI o mediante un *Shell*.
- Adopción de procesos huérfanos: Cuando un proceso inicia un proceso hijo y éste muere antes que el padre, el proceso hijo pasa a ser un hijo de *init* momentáneamente. También adopta procesos hijos si un proceso padre termina sin esperarlos, liberando su ocupación en memoria principal.
- Al cerrar el sistema, es *init* quien se encarga de matar todos los procesos restantes, desmontar todos los sistemas de archivos y de cualquier otra cosa que haya sido configurado para hacer.

Entre los procesos generados por *init* están los procesos *getty*. Se arrancará un proceso *getty* por cada terminal. Este proceso se transforma con *exec()* en el proceso *login*. Para ello se invoca al ejecutable “/bin/login”, que consulta las credenciales del usuario en “/etc/passwd”, junto con el nivel de privilegios y tipo de *Shell* utilizado. El proceso *login* a su vez se transforma con un *exec()*

en una *Shell*. Por último, cada *Shell* espera por un comando y luego lanza un *fork()* y un *exec()* por cada comando introducido correctamente.



## 13 Un poco de historia



## 1.2 HISTORIA DE LOS SISTEMAS OPERATIVOS

Los sistemas operativos han ido evolucionando a través de los años. En las siguientes secciones analizaremos brevemente algunos de los hitos más importantes. Como los sistemas operativos han estado estrechamente relacionados a través de la historia con la arquitectura de las computadoras en las que se ejecutan, analizaremos generaciones sucesivas de computadoras para ver cómo eran sus sistemas operativos. Esta vinculación de generaciones de sistemas operativos con generaciones de computadoras es un poco burda, pero proporciona cierta estructura donde de cualquier otra forma no habría.

La progresión que se muestra a continuación es en gran parte cronológica, aunque el desarrollo ha sido un tanto accidentado. Cada fase surgió sin esperar a que la anterior terminara completamente. Hubo muchos traslapes, sin mencionar muchos falsos inicios y callejones sin salida. El lector debe tomar esto como guía, no como la última palabra.

La primera computadora digital verdadera fue diseñada por el matemático inglés Charles Babbage (de 1792 a 1871). Aunque Babbage gastó la mayor parte de su vida y fortuna tratando de construir su “máquina analítica”, nunca logró hacer que funcionara de manera apropiada, debido a que era puramente mecánica y la tecnología de su era no podía producir las ruedas, engranes y dientes con la alta precisión que requería. Por supuesto, la máquina analítica no tenía un sistema operativo.

Como nota histórica interesante, Babbage se dio cuenta de que necesitaba software para su máquina analítica, por lo cual contrató a una joven llamada Ada Lovelace, hija del afamado poeta británico Lord Byron, como la primera programadora del mundo. El lenguaje de programación Ada® lleva su nombre.

### 1.2.1 La primera generación (1945 a 1955): tubos al vacío

Después de los esfuerzos infructuosos de Babbage, no hubo muchos progresos en la construcción de computadoras digitales sino hasta la Segunda Guerra Mundial, que estimuló una explosión de esta actividad. El profesor John Atanasoff y su estudiante graduado Clifford Berry construyeron lo que ahora se conoce como la primera computadora digital funcional en Iowa State University. Utilizaba 300 tubos de vacío (bulbos). Aproximadamente al mismo tiempo, Konrad Zuse en Berlín

construyó la computadora Z3 a partir de relevadores. En 1944, la máquina Colossus fue construida por un equipo de trabajo en Bletchley Park, Inglaterra; la Mark I, por Howard Aiken en Harvard, y la ENIAC, por William Mauchley y su estudiante graduado J. Presper Eckert en la Universidad de Pennsylvania. Algunas fueron binarias, otras utilizaron bulbos, algunas eran programables, pero todas eran muy primitivas y tardaban segundos en realizar incluso hasta el cálculo más simple.

En estos primeros días, un solo grupo de personas (generalmente ingenieros) diseñaban, construían, programaban, operaban y daban mantenimiento a cada máquina. Toda la programación se realizaba exclusivamente en lenguaje máquina o, peor aún, creando circuitos eléctricos mediante la conexión de miles de cables a tableros de conexiones (*plugboards*) para controlar las funciones básicas de la máquina. Los lenguajes de programación eran desconocidos (incluso se desconocía el lenguaje ensamblador). Los sistemas operativos también se desconocían. El modo usual de operación consistía en que el programador trabajaba un periodo dado, registrándose en una hoja de firmas, y después entraba al cuarto de máquinas, insertaba su tablero de conexiones en la computadora e invertía varias horas esperando que ninguno de los cerca de 20,000 bulbos se quemara durante la ejecución. Prácticamente todos los problemas eran cálculos numéricos bastante simples, como obtener tablas de senos, cosenos y logaritmos.

A principios de la década de 1950, la rutina había mejorado un poco con la introducción de las tarjetas perforadas. Entonces fue posible escribir programas en tarjetas y leerlos en vez de usar tableros de conexiones; aparte de esto, el procedimiento era el mismo.

### 1.2.2 La segunda generación (1955 a 1965): transistores y sistemas de procesamiento por lotes

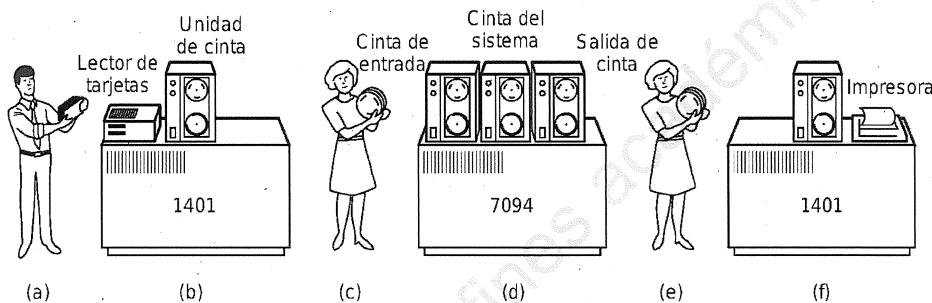
La introducción del transistor a mediados de la década de 1950 cambió radicalmente el panorama. Las computadoras se volvieron lo bastante confiables como para poder fabricarlas y venderlas a clientes dispuestos a pagar por ellas, con la expectativa de que seguirían funcionando el tiempo suficiente como para poder llevar a cabo una cantidad útil de trabajo. Por primera vez había una clara separación entre los diseñadores, constructores, operadores, programadores y el personal de mantenimiento.

Estas máquinas, ahora conocidas como **mainframes**, estaban encerradas en cuartos especiales con aire acondicionado y grupos de operadores profesionales para manejarlas. Sólo las empresas grandes, universidades o agencias gubernamentales importantes podían financiar el costo multimillonario de operar estas máquinas. Para ejecutar un **trabajo** (es decir, un programa o conjunto de programas), el programador primero escribía el programa en papel (en FORTRAN o en ensamblador) y después lo pasaba a tarjetas perforadas. Luego llevaba el conjunto de tarjetas al cuarto de entrada de datos y lo entregaba a uno de los operadores; después se iba a tomar un café a esperar a que los resultados estuvieran listos.

Cuando la computadora terminaba el trabajo que estaba ejecutando en un momento dado, un operador iba a la impresora y arrancaba las hojas de resultados para llevarlas al cuarto de salida de datos, para que el programador pudiera recogerlas posteriormente. Entonces, el operador tomaba uno de los conjuntos de tarjetas que se habían traído del cuarto de entrada y las introducía en la máquina. Si se necesitaba el compilador FORTRAN, el operador tenía que obtenerlo de un gabinete

de archivos e introducirlo a la máquina. Se desperdiciaba mucho tiempo de la computadora mientras los operadores caminaban de un lado a otro del cuarto de la máquina.

Dado el alto costo del equipo, no es sorprendente que las personas buscaran rápidamente formas de reducir el tiempo desperdiciado. La solución que se adoptó en forma general fue el **sistema de procesamiento por lotes**. La idea detrás de este concepto era recolectar una bandeja llena de trabajos en el cuarto de entrada de datos y luego pasarlo a una cinta magnética mediante el uso de una pequeña computadora relativamente económica, tal como la IBM 1401, que era muy adecuada para leer las tarjetas, copiar cintas e imprimir los resultados, pero no tan buena para los cálculos numéricos. Para llevar a cabo los cálculos numéricos se utilizaron otras máquinas mucho más costosas, como la IBM 7094. Este procedimiento se ilustra en la figura 1-3.

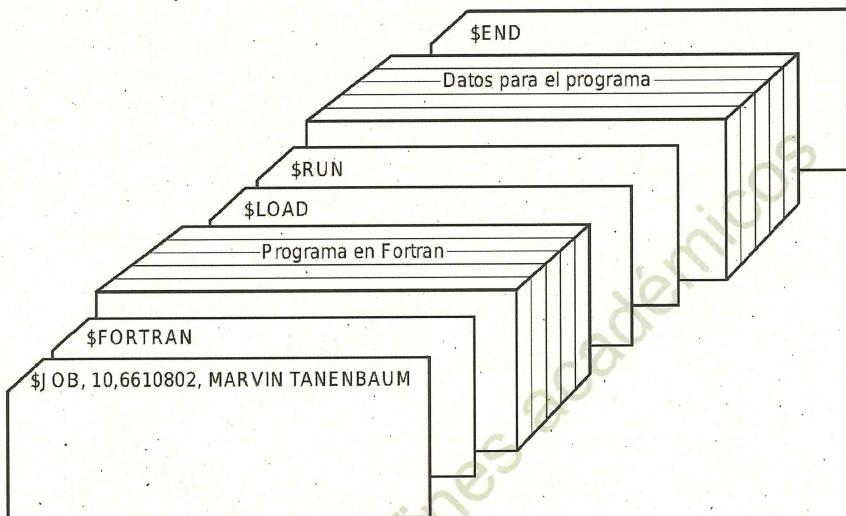


**Figura 1-3.** Uno de los primeros sistemas de procesamiento por lotes. a) Los programadores llevan las tarjetas a la 1401. b) La 1401 lee los lotes de trabajos y los coloca en cinta. c) El operador lleva la cinta de entrada a la 7094. d) La 7094 realiza los cálculos. e) El operador lleva la cinta de salida a la 1401. f) La 1401 imprime los resultados.

Después de aproximadamente una hora de recolectar un lote de trabajos, las tarjetas se leían y se colocaban en una cinta magnética, la cual se llevaba al cuarto de máquinas, en donde se montaba en una unidad de cinta. Después, el operador cargaba un programa especial (el ancestro del sistema operativo de hoy en día), el cual leía el primer trabajo de la cinta y lo ejecutaba. Los resultados se escribían en una segunda cinta, en vez de imprimirlos. Después de que terminaba cada trabajo, el sistema operativo leía de manera automática el siguiente trabajo de la cinta y empezaba a ejecutarlo. Cuando se terminaba de ejecutar todo el lote, el operador quitaba las cintas de entrada y de salida, reemplazaba la cinta de entrada con el siguiente lote y llevaba la cinta de salida a una 1401 para imprimir **fuerza de línea** (es decir, sin conexión con la computadora principal).

En la figura 1-4 se muestra la estructura típica de un trabajo de entrada ordinario. Empieza con una tarjeta \$JOB, especificando el tiempo máximo de ejecución en minutos, el número de cuenta al que se va a cargar y el nombre del programador. Después se utiliza una tarjeta \$FORTRAN, indicando al sistema operativo que debe cargar el compilador FORTRAN de la cinta del sistema. Después le sigue inmediatamente el programa que se va a compilar y luego una tarjeta \$LOAD, que indica al sistema operativo que debe cargar el programa objeto que acaba de compilar (a menudo, los programas compilados se escribían en cintas reutilizables y tenían que cargarse en forma explícita). Después se utiliza la tarjeta \$RUN, la cual indica al sistema operativo que debe ejecutar el

programa con los datos que le suceden. Por último, la tarjeta \$END marca el final del trabajo. Esas tarjetas de control primitivas fueron las precursoras de los shells e intérpretes de línea de comandos modernos.



**Figura 1-4.** Estructura de un trabajo típico de FMS.

Las computadoras grandes de segunda generación se utilizaron principalmente para cálculos científicos y de ingeniería, tales como resolver ecuaciones diferenciales parciales que surgen a menudo en física e ingeniería. En gran parte se programaron en FORTRAN y lenguaje ensamblador. Los sistemas operativos típicos eran FMS (Fortran Monitor System) e IBSYS, el sistema operativo de IBM para la 7094.

### 1.2.3 La tercera generación (1965 a 1980): circuitos integrados y multiprogramación

A principio de la década de 1960, la mayoría de los fabricantes de computadoras tenían dos líneas de productos distintas e incompatibles. Por una parte estaban las computadoras científicas a gran escala orientadas a palabras, como la 7094, que se utilizaban para cálculos numéricos en ciencia e ingeniería. Por otro lado, estaban las computadoras comerciales orientadas a caracteres, como la 1401, que se utilizaban ampliamente para ordenar cintas e imprimir datos en los bancos y las compañías de seguros.

Desarrollar y dar mantenimiento a dos líneas de productos completamente distintos era una propuesta costosa para los fabricantes. Además, muchos nuevos clientes de computadoras necesitaban al principio un equipo pequeño, pero más adelante ya no era suficiente y deseaban una máquina más grande que pudiera ejecutar todos sus programas anteriores, pero con mayor rapidez.

IBM intentó resolver ambos problemas de un solo golpe con la introducción de la línea de computadoras System/360. La 360 era una serie de máquinas compatibles con el software, que variaban desde un tamaño similar a la 1401 hasta algunas que eran más potentes que la 7094. Las máquinas sólo diferían en el precio y rendimiento (máxima memoria, velocidad del procesador, número de dispositivos de E/S permitidos, etcétera). Como todas las máquinas tenían la misma arquitectura y el mismo conjunto de instrucciones, los programas escritos para una máquina podían ejecutarse en todas las demás, por lo menos en teoría. Lo que es más, la 360 se diseñó para manejar tanto la computación científica (es decir, numérica) como comercial. Por ende, una sola familia de máquinas podía satisfacer las necesidades de todos los clientes. En los años siguientes, mediante el uso de tecnología más moderna, IBM ha desarrollado sucesores compatibles con la línea 360, a los cuales se les conoce como modelos 370, 4300, 3080 y 3090. La serie zSeries es el descendiente más reciente de esta línea, aunque diverge considerablemente del original.

La IBM 360 fue la primera línea importante de computadoras en utilizar **circuitos integrados (ICs)** (a pequeña escala), con lo cual se pudo ofrecer una mayor ventaja de precio/rendimiento en comparación con las máquinas de segunda generación, las cuales fueron construidas a partir de transistores individuales. Su éxito fue inmediato y la idea de una familia de computadoras compatibles pronto fue adoptada por todos los demás fabricantes importantes. Los descendientes de estas máquinas se siguen utilizando hoy día en centros de cómputo. En la actualidad se utilizan con frecuencia para manejar bases de datos enormes (por ejemplo, para sistemas de reservaciones de aerolíneas) o como servidores para sitios de World Wide Web que deben procesar miles de solicitudes por segundo.

La mayor fortaleza de la idea de “una sola familia” fue al mismo tiempo su mayor debilidad. La intención era que todo el software, incluyendo al sistema operativo **OS/360**, funcionara en todos los modelos. Debía ejecutarse en los sistemas pequeños, que por lo general sólo reemplazaban a la 1401s, que copiaba tarjetas a cinta, y en los sistemas muy grandes, que a menudo reemplazaban a la 7094s, que realizaba predicciones del clima y otros cálculos pesados. Tenía que ser bueno en sistemas con pocos dispositivos periféricos y en sistemas con muchos. Tenía que funcionar en ambos entornos comerciales y científicos. Por encima de todo, tenía que ser eficiente para todos estos usos distintos.

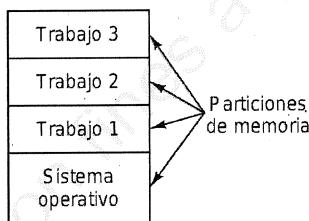
No había forma en que IBM (o cualquier otra) pudiera escribir una pieza de software que cumpliera con todos estos requerimientos en conflicto. El resultado fue un enorme y extraordinariamente complejo sistema operativo, tal vez de dos a tres órdenes de magnitud más grande que el FMS. Consistía en millones de líneas de lenguaje ensamblador escrito por miles de programadores, con miles de errores, los cuales requerían un flujo continuo de nuevas versiones en un intento por corregirlos. Cada nueva versión corría algunos errores e introducía otros, por lo que probablemente el número de errores permanecía constante en el tiempo.

Fred Brooks, uno de los diseñadores del OS/360, escribió posteriormente un libro ingenioso e incisivo (Brooks, 1996) que describía sus experiencias con el OS/360. Aunque sería imposible resumir este libro aquí, basta con decir que la portada muestra una manada de bestias prehistóricas atrapadas en un pozo de brea. La portada de Silberschatz y coautores (2005) muestra un punto de vista similar acerca de que los sistemas operativos son como dinosaurios.

A pesar de su enorme tamaño y sus problemas, el OS/360 y los sistemas operativos similares de tercera generación producidos por otros fabricantes de computadoras en realidad dejaban razon-

nablemente satisfechos a la mayoría de sus clientes. También popularizaron varias técnicas clave ausentes en los sistemas operativos de segunda generación. Quizá la más importante de éstas fue la **multiprogramación**. En la 7094, cuando el trabajo actual se detenía para esperar a que se completara una operación con cinta u otro dispositivo de E/S, la CPU simplemente permanecía inactiva hasta terminar la operación de E/S. Con los cálculos científicos que requieren un uso intensivo de la CPU, la E/S no es frecuente, por lo que este tiempo desperdiciado no es considerable. Con el procesamiento de datos comerciales, el tiempo de espera de las operaciones de E/S puede ser a menudo de 80 a 90 por ciento del tiempo total, por lo que debía hacerse algo para evitar que la (costosa) CPU esté inactiva por mucho tiempo.

La solución que surgió fue partitionar la memoria en varias piezas, con un trabajo distinto en cada partición, como se muestra en la figura 1-5. Mientras que un trabajo esperaba a que se completara una operación de E/S, otro podía estar usando la CPU. Si pudieran contenerse suficientes trabajos en memoria principal al mismo tiempo, la CPU podía estar ocupada casi 100 por ciento del tiempo. Para tener varios trabajos de forma segura en memoria a la vez, se requiere hardware especial para proteger cada trabajo y evitar que los otros se entrometan y lo malogren; el 360 y los demás sistemas de tercera generación estaban equipados con este hardware.



**Figura 1-5.** Un sistema de multiprogramación con tres trabajos en memoria.

Otra característica importante de los sistemas operativos de tercera generación fue la capacidad para leer trabajos en tarjetas y colocarlos en el disco tan pronto como se llevaban al cuarto de computadoras. Así, cada vez que terminaba un trabajo en ejecución, el sistema operativo podía cargar un nuevo trabajo del disco en la partición que entonces estaba vacía y lo ejecutaba. A esta técnica se le conoce como **spooling** (de *Simultaneous Peripheral Operation On Line*, operación periférica simultánea en línea) y también se utilizó para las operaciones de salida. Con el spooling, las máquinas 1401 no eran ya necesarias y desapareció la mayor parte del trabajo de transportar las cintas.

Aunque los sistemas operativos de tercera generación eran apropiados para los cálculos científicos extensos y las ejecuciones de procesamiento de datos comerciales masivos, seguían siendo en esencia sistemas de procesamiento por lotes. Muchos programadores añoraban los días de la primera generación en los que tenían toda la máquina para ellos durante unas cuantas horas, por lo que podían depurar sus programas con rapidez. Con los sistemas de tercera generación, el tiempo que transcurría entre enviar un trabajo y recibir de vuelta la salida era comúnmente de varias horas, por lo que una sola coma mal colocada podía ocasionar que fallara la compilación, y el programador perdería la mitad del día.

Este deseo de obtener un tiempo rápido de respuesta allanó el camino para el **tiempo compartido** (*timesharing*), una variante de la multiprogramación donde cada usuario tenía una terminal en

línea. En un sistema de tiempo compartido, si 20 usuarios están conectados y 17 de ellos están pensando en dar un paseo o tomar café, la CPU se puede asignar por turno a los tres trabajos que desean ser atendidos. Como las personas que depuran programas generalmente envían comandos cortos (por ejemplo, compilar un procedimiento de cinco hojas<sup>†</sup>) en vez de largos (por ejemplo, ordenar un archivo con un millón de registros), la computadora puede proporcionar un servicio rápido e interactivo a varios usuarios y, tal vez, también ocuparse en trabajos grandes por lotes en segundo plano, cuando la CPU estaría inactiva de otra manera. El primer sistema de tiempo compartido de propósito general, conocido como **CTSS** (*Compatible Time Sharing System*, Sistema compatible de tiempo compartido), se desarrolló en el M.I.T. en una 7094 modificada en forma especial (Corbató y colaboradores, 1962). Sin embargo, en realidad el tiempo compartido no se popularizó sino hasta que el hardware de protección necesario se empezó a utilizar ampliamente durante la tercera generación.

Después del éxito del sistema CTSS, el M.I.T., Bell Labs y General Electric (que en ese entonces era un importante fabricante de computadoras) decidieron emprender el desarrollo de una “utilería para computadora”, una máquina capaz de servir a varios cientos de usuarios simultáneos de tiempo compartido. Su modelo fue el sistema de electricidad: cuando se necesita energía, sólo hay que conectar un contacto a la pared y, dentro de lo razonable, toda la energía que se requiera estará ahí. Los diseñadores del sistema conocido como **MULTICS** (*MULTplexed Information and Computing Service*; Servicio de Información y Cómputo MULTplexado), imaginaron una enorme máquina que proporcionaba poder de cómputo a todos los usuarios en el área de Boston. La idea de que, sólo 40 años después, se vendieran por millones máquinas 10,000 veces más rápidas que su mainframe GE-645 (a un precio muy por debajo de los 1000 dólares) era pura ciencia ficción. Algo así como la idea de que en estos días existiera un transatlántico supersónico por debajo del agua.

MULTICS fue un éxito parcial. Se diseñó para dar soporte a cientos de usuarios en una máquina que era sólo un poco más potente que una PC basada en el Intel 386, aunque tenía mucho más capacidad de E/S. Esto no es tan disparatado como parece, ya que las personas sabían cómo escribir programas pequeños y eficientes en esos días, una habilidad que se ha perdido con el tiempo. Hubo muchas razones por las que MULTICS no acaparó la atención mundial; una de ellas fue el que estaba escrito en PL/I y el compilador de PL/I se demoró por años, además de que apenas funcionaba cuando por fin llegó. Aparte de eso, MULTICS era un sistema demasiado ambicioso para su época, algo muy parecido a la máquina analítica de Charles Babbage en el siglo diecinueve.

Para resumir esta larga historia, MULTICS introdujo muchas ideas seminales en la literatura de las computadoras, pero convertirlas en un producto serio y con éxito comercial importante era algo mucho más difícil de lo que cualquiera hubiera esperado. Bell Labs se retiró del proyecto y General Electric dejó el negocio de las computadoras por completo. Sin embargo, el M.I.T. persistió y logró hacer en un momento dado que MULTICS funcionara. Al final, la compañía que compró el negocio de computadoras de GE (Honeywell) lo vendió como un producto comercial y fue instalado por cerca de 80 compañías y universidades importantes a nivel mundial. Aunque en número pequeño, los usuarios de MULTICS eran muy leales. Por ejemplo, General Motors, Ford y la Agencia de Seguridad Nacional de los Estados Unidos desconectaron sus sistemas MULTICS sólo hasta

<sup>†</sup> En este libro utilizaremos los términos “procedimiento”, “subrutina” y “función” de manera indistinta.

finales de la década de 1990, 30 años después de su presentación en el mercado y de tratar durante años de hacer que Honeywell actualizara el hardware.

Por ahora, el concepto de una “utilería para computadora” se ha disipado, pero tal vez regrese en forma de servidores masivos de Internet centralizados a los que se conecten máquinas de usuario relativamente “tontas”, donde la mayoría del trabajo se realice en los servidores grandes. Es probable que la motivación en este caso sea que la mayoría de las personas no desean administrar un sistema de cómputo cada vez más complejo y melindroso, y prefieren delegar esa tarea a un equipo de profesionales que trabajen para la compañía que opera el servidor. El comercio electrónico ya está evolucionando en esta dirección, en donde varias compañías operan centros comerciales electrónicos en servidores multiprocesador a los que se conectan las máquinas cliente simples, algo muy parecido al diseño de MULTICS.

A pesar de la carencia de éxito comercial, MULTICS tuvo una enorme influencia en los sistemas operativos subsecuentes. Se describe en varios artículos y en un libro (Corbató y colaboradores, 1972; Corbató y Vyssotsky, 1965; Daley y Dennis, 1968; Organick, 1972; y Stoltzer, 1974). También tuvo (y aún tiene) un sitio Web activo, ubicado en [www.multicians.org](http://www.multicians.org), con mucha información acerca del sistema, sus diseñadores y sus usuarios.

Otro desarrollo importante durante la tercera generación fue el increíble crecimiento de las minicomputadoras, empezando con la DEC PDP-1 en 1961. La PDP-1 tenía sólo 4K de palabras de 18 bits, pero a \$120,000 por máquina (menos de 5 por ciento del precio de una 7094) se vendió como pan caliente. Para cierta clase de trabajo no numérico, era casi tan rápida como la 7094 y dio origen a una nueva industria. A esta minicomputadora le siguió rápidamente una serie de otras PDP (a diferencia de la familia de IBM, todas eran incompatibles), culminando con la PDP-11.

Posteriormente, Ken Thompson, uno de los científicos de cómputo en Bell Labs que trabajó en el proyecto MULTICS, encontró una pequeña minicomputadora PDP-7 que nadie estaba usando y se dispuso a escribir una versión simple de MULTICS para un solo usuario. Más adelante, este trabajo se convirtió en el sistema operativo **UNIX®**, que se hizo popular en el mundo académico, las agencias gubernamentales y muchas compañías.

La historia de UNIX ya ha sido contada en muchos otros libros (por ejemplo, Salus, 1994). En el capítulo 10 hablaremos sobre parte de esa historia. Por ahora baste con decir que, debido a que el código fuente estaba disponible ampliamente, varias organizaciones desarrollaron sus propias versiones (incompatibles entre sí), lo cual produjo un caos. Se desarrollaron dos versiones principales: **System V** de AT&T y **BSD** (*Berkeley Software Distribution*, Distribución de Software de Berkeley) de la Universidad de California en Berkeley. Estas versiones tenían también variantes menores. Para que fuera posible escribir programas que pudieran ejecutarse en cualquier sistema UNIX, el IEEE desarrolló un estándar para UNIX conocido como **POSIX**, con el que la mayoría de las versiones de UNIX actuales cumplen. POSIX define una interfaz mínima de llamadas al sistema a la que los sistemas UNIX deben conformarse. De hecho, algunos de los otros sistemas operativos también admiten ahora la interfaz POSIX.

Como agregado, vale la pena mencionar que en 1987 el autor liberó un pequeño clon de UNIX conocido como **MINIX**, con fines educativos. En cuanto a su funcionalidad, MINIX es muy similar a UNIX, incluyendo el soporte para POSIX. Desde esa época, la versión original ha evolucionado en MINIX 3, que es altamente modular y está enfocada a presentar una muy alta confiabilidad. Tiene la habilidad de detectar y reemplazar módulos con fallas o incluso inutilizables (como los dis-

positivos controladores de dispositivos de E/S) al instante, sin necesidad de reiniciar y sin perturbar a los programas en ejecución. También hay disponible un libro que describe su operación interna y contiene un listado del código fuente en un apéndice (Tanenbaum y Woodhull, 2006). El sistema MINIX 3 está disponible en forma gratuita (incluyendo todo el código fuente) a través de Internet, en [www.minix3.org](http://www.minix3.org).

El deseo de una versión de producción (en vez de educativa) gratuita de MINIX llevó a un estudiante finlandés, llamado Linus Torvalds, a escribir **Linux**. Este sistema estaba inspirado por MINIX, además de que fue desarrollado en este sistema y originalmente ofrecía soporte para varias características de MINIX (por ejemplo, el sistema de archivos de MINIX). Desde entonces se ha extendido en muchas formas, pero todavía retiene cierta parte de su estructura subyacente común para MINIX y UNIX. Los lectores interesados en una historia detallada sobre Linux y el movimiento de código fuente abierto tal vez deseen leer el libro de Glyn Moody (2001). La mayor parte de lo que se haya dicho acerca de UNIX en este libro se aplica también a System V, MINIX, Linux y otras versiones y clones de UNIX.

#### 1.2.4 La cuarta generación (1980 a la fecha): las computadoras personales

Con el desarrollo de los circuitos LSI (*Large Scale Integration*, Integración a gran escala), que contienen miles de transistores en un centímetro cuadrado de silicio (chip), nació la era de la computadora personal. En términos de arquitectura, las computadoras personales (que al principio eran conocidas como **microcomputadoras**) no eran del todo distintas de las minicomputadoras de la clase PDP-11, pero en términos de precio sin duda eran distintas. Mientras que la minicomputadora hizo posible que un departamento en una compañía o universidad tuviera su propia computadora, el chip microprocesador logró que un individuo tuviera su propia computadora personal.

Cuando Intel presentó el microprocesador 8080 en 1974 (la primera CPU de 8 bits de propósito general), deseaba un sistema operativo, en parte para poder probarlo. Intel pidió a uno de sus consultores, Gary Kildall, que escribiera uno. Kildall y un amigo construyeron primero un dispositivo controlador para el disco flexible de 8 pulgadas de Shugart Associates que recién había sido sacado al mercado, y conectaron el disco flexible con el 8080, con lo cual produjeron la primera microcomputadora con un disco. Después Kildall escribió un sistema operativo basado en disco conocido como **CP/M** (*Control Program for Microcomputers*; Programa de Control para Microcomputadoras) para esta CPU. Como Intel no pensó que las microcomputadoras basadas en disco tuvieran mucho futuro, cuando Kildall pidió los derechos para CP/M, Intel le concedió su petición. Después Kildall formó una compañía llamada Digital Research para desarrollar y vender el CP/M.

En 1977, Digital Research rediseñó el CP/M para adaptarlo de manera que se pudiera ejecutar en todas las microcomputadoras que utilizaban los chips 8080, Zilog Z80 y otros. Se escribieron muchos programas de aplicación para ejecutarse en CP/M, lo cual le permitió dominar por completo el mundo de la microcomputación durante un tiempo aproximado de 5 años.

A principios de la década de 1980, IBM diseñó la IBM PC y buscó software para ejecutarlo en ella. La gente de IBM se puso en contacto con Bill Gates para obtener una licencia de uso de su intérprete de BASIC. También le preguntaron si sabía de un sistema operativo que se ejecutara en la PC. Gates sugirió a IBM que se pusiera en contacto con Digital Research, que en ese entonces era la compañía con dominio mundial en el área de sistemas operativos. Kildall rehusó a reunirse con IBM y envió a uno de sus subordinados, a lo cual se le considera sin duda la peor decisión de negocios de la historia. Para empeorar más aún las cosas, su abogado se rehusó a firmar el contrato de no divulgación de IBM sobre la PC, que no se había anunciado todavía. IBM regresó con Gates para ver si podía proveerles un sistema operativo.

Cuando IBM regresó, Gates se había enterado de que un fabricante local de computadoras, Seattle Computer Products, tenía un sistema operativo adecuado conocido como **DOS** (*Disk Operating System*; Sistema Operativo en Disco). Se acercó a ellos y les ofreció comprarlo (supuestamente por 75,000 dólares), a lo cual ellos accedieron de buena manera. Después Gates ofreció a IBM un paquete con DOS/BASIC, el cual aceptó. IBM quería ciertas modificaciones, por lo que Gates contrató a la persona que escribió el DOS, Tim Paterson, como empleado de su recién creada empresa de nombre Microsoft, para que las llevara a cabo. El sistema rediseñado cambió su nombre a **MS-DOS** (*Microsoft Disk Operating System*; Sistema Operativo en Disco de Microsoft) y rápidamente llegó a dominar el mercado de la IBM PC. Un factor clave aquí fue la decisión de Gates (que en retrospectiva, fue en extremo inteligente) de vender MS-DOS a las empresas de computadoras para que lo incluyeran con su hardware, en comparación con el intento de Kildall por vender CP/M a los usuarios finales, uno a la vez (por lo menos al principio). Después de que se supo todo esto, Kildall murió en forma repentina e inesperada debido a causas que aún no han sido reveladas por completo.

Para cuando salió al mercado en 1983 la IBM PC/AT, sucesora de la IBM PC, con la CPU Intel 80286, MS-DOS estaba muy afianzado y CP/M daba sus últimos suspiros. Más adelante, MS-DOS se utilizó ampliamente en el 80386 y 80486. Aunque la versión inicial de MS-DOS era bastante primitiva, las versiones siguientes tenían características más avanzadas, incluyendo muchas que se tomaron de UNIX. (Microsoft estaba muy al tanto de UNIX e inclusive vendía una versión de este sistema para microcomputadora, conocida como XENIX, durante los primeros años de la compañía).

CP/M, MS-DOS y otros sistemas operativos para las primeras microcomputadoras se basaban en que los usuarios escribieran los comandos mediante el teclado. Con el tiempo esto cambió debido a la investigación realizada por Doug Engelbart en el Stanford Research Institute en la década de 1960. Engelbart inventó la **Interfaz Gráfica de Usuario GUI**, completa con ventanas, iconos, menús y ratón. Los investigadores en Xerox PARC adoptaron estas ideas y las incorporaron en las máquinas que construyeron.

Un día, Steve Jobs, que fue co-inventor de la computadora Apple en su cochera, visitó PARC, vio una GUI y de inmediato se dio cuenta de su valor potencial, algo que la administración de Xerox no hizo. Esta equivocación estratégica de gigantescas proporciones condujo a un libro titulado *Fumbling the Future* (Smith y Alexander, 1988). Posteriormente, Jobs emprendió el proyecto de construir una Apple con una GUI. Este proyecto culminó en Lisa, que era demasiado costosa y fracasó comercialmente. El segundo intento de Jobs, la Apple Macintosh, fue un enorme éxito, no sólo debido a que era mucho más económica que Lisa, sino también porque era **amigable para el**

**usuario** (*user friendly*), lo cual significaba que estaba diseñada para los usuarios que no sólo no sabían nada acerca de las computadoras, sino que además no tenían ninguna intención de aprender. En el mundo creativo del diseño gráfico, la fotografía digital profesional y la producción de video digital profesional, las Macintosh son ampliamente utilizadas y sus usuarios son muy entusiastas sobre ellas.

Cuando Microsoft decidió crear un sucesor para el MS-DOS estaba fuertemente influenciado por el éxito de la Macintosh. Produjo un sistema basado en GUI llamado Windows, el cual en un principio se ejecutaba encima del MS-DOS (es decir, era más como un shell que un verdadero sistema operativo). Durante cerca de 10 años, de 1985 a 1995, Windows fue sólo un entorno gráfico encima de MS-DOS. Sin embargo, a partir de 1995 se liberó una versión independiente de Windows, conocida como Windows 95, que incorporaba muchas características de los sistemas operativos y utilizaba el sistema MS-DOS subyacente sólo para iniciar y ejecutar programas de MS-DOS antiguos. En 1998, se liberó una versión ligeramente modificada de este sistema, conocida como Windows 98. Sin embargo, tanto Windows 95 como Windows 98 aún contenían una gran cantidad de lenguaje ensamblador para los procesadores Intel de 16 bits.

Otro de los sistemas operativos de Microsoft es **Windows NT** (NT significa Nueva Tecnología), que es compatible con Windows 95 en cierto nivel, pero fue completamente rediseñado en su interior. Es un sistema completo de 32 bits. El diseñador en jefe de Windows NT fue David Cutler, quien también fue uno de los diseñadores del sistema operativo VMS de VAX, por lo que hay algunas ideas de VMS presentes en NT. De hecho, había tantas ideas de VMS presentes que el propietario de VMS (DEC) demandó a Microsoft. El caso se resolvió en la corte por una cantidad de muchos dígitos. Microsoft esperaba que la primera versión de NT acabara con MS-DOS y todas las demás versiones de Windows, ya que era un sistema muy superior, pero fracasó. No fue sino hasta Windows NT 4.0 que finalmente empezó a tener éxito, en especial en las redes corporativas. La versión 5 de Windows NT cambió su nombre a Windows 2000 a principios de 1999. Estaba destinada a ser el sucesor de Windows 98 y de Windows NT 4.0.

Esto tampoco funcionó como se esperaba, por lo que Microsoft preparó otra versión de Windows 98 conocida como **Windows Me** (*Millennium edition*). En el 2001 se liberó una versión ligeramente actualizada de Windows 2000, conocida como Windows XP. Esta versión duró mucho más en el mercado (6 años), reemplazando a casi todas las versiones anteriores de Windows. Después, en enero del 2007 Microsoft liberó el sucesor para Windows XP, conocido como Windows Vista. Tenía una interfaz gráfica nueva, Aero, y muchos programas de usuario nuevos o actualizados. Microsoft esperaba que sustituya a Windows XP por completo, pero este proceso podría durar casi toda una década.

El otro competidor importante en el mundo de las computadoras personales es UNIX (y todas sus variantes). UNIX es más fuerte en los servidores tanto de redes como empresariales, pero también está cada vez más presente en las computadoras de escritorio, en especial en los países que se desarrollan con rapidez, como India y China. En las computadoras basadas en Pentium, Linux se está convirtiendo en una alternativa popular para Windows entre los estudiantes y cada vez más usuarios corporativos. Como agregado, a lo largo de este libro utilizaremos el término “Pentium” para denotar al Pentium I, II, III y 4, así como sus sucesores tales como el Core 2 Duo. El término **x86** también se utiliza algunas veces para indicar el rango completo de CPU Intel partiendo desde el 8086, mientras que utilizaremos “Pentium” para indicar todas las CPU desde el

Pentium I. Admitimos que este término no es perfecto, pero no hay disponible uno mejor. Uno se pregunta qué genio de mercadotecnia en Intel desperdició una marca comercial (Pentium) que la mitad del mundo conocía bien y respetaba, sustituyéndola con términos como “Core 2 duo” que muy pocas personas comprenden; ¿qué significan “2” y “duo”? Tal vez “Pentium 5” (o “Pentium 5 dual core”, etc.) eran demasiado difíciles de recordar. **FreeBSD** es también un derivado popular de UNIX, que se originó del proyecto BSD en Berkeley. Todas las computadoras modernas Macintosh utilizan una versión modificada de FreeBSD. UNIX también es estándar en las estaciones de trabajo operadas por chips RISC de alto rendimiento, como los que venden Hewlett-Packard y Sun Microsystems.

Muchos usuarios de UNIX, en especial los programadores experimentados, prefieren una interfaz de línea de comandos a una GUI, por lo que casi todos los sistemas UNIX presentan un sistema de ventanas llamado **X Window System** (también conocido como **X11**), producido en el M.I.T. Este sistema se encarga de la administración básica de las ventanas y permite a los usuarios crear, eliminar, desplazar y cambiar el tamaño de las ventanas mediante el uso de un ratón. Con frecuencia hay disponible una GUI completa, como **Gnome** o **KDE**, para ejecutarse encima de X11, lo cual proporciona a UNIX una apariencia parecida a la Macintosh o a Microsoft Windows, para aquellos usuarios de UNIX que desean algo así.

Un interesante desarrollo que empezó a surgir a mediados de la década de 1980 es el crecimiento de las redes de computadoras personales que ejecutan **sistemas operativos en red** y **sistemas operativos distribuidos** (Tanenbaum y Van Steen, 2007). En un sistema operativo en red, los usuarios están conscientes de la existencia de varias computadoras, y pueden iniciar sesión en equipos remotos y copiar archivos de un equipo a otro. Cada equipo ejecuta su propio sistema operativo local y tiene su propio usuario (o usuarios) local.

Los sistemas operativos en red no son fundamentalmente distintos de los sistemas operativos con un solo procesador. Es obvio que necesitan un dispositivo controlador de interfaz de red y cierto software de bajo nivel para controlarlo, así como programas para lograr el inicio de una sesión remota y el acceso remoto a los archivos, pero estas adiciones no cambian la estructura esencial del sistema operativo.

En contraste, un sistema operativo distribuido se presenta a sus usuarios en forma de un sistema tradicional con un procesador, aun cuando en realidad está compuesto de varios procesadores. Los usuarios no tienen que saber en dónde se están ejecutando sus programas o en dónde se encuentran sus archivos; el sistema operativo se encarga de todo esto de manera automática y eficiente.

Los verdaderos sistemas operativos distribuidos requieren algo más que sólo agregar un poco de código a un sistema operativo con un solo procesador, ya que los sistemas distribuidos y los centralizados difieren en varios puntos críticos. Por ejemplo, los sistemas distribuidos permiten con frecuencia que las aplicaciones se ejecuten en varios procesadores al mismo tiempo, lo que requiere algoritmos de planificación del procesador más complejos para poder optimizar la cantidad de paralelismo.

Los retrasos de comunicación dentro de la red implican a menudo que estos (y otros) algoritmos deben ejecutarse con información incompleta, obsoleta o incluso incorrecta. Esta situación es muy distinta a la de un sistema con un solo procesador, donde el sistema operativo tiene información completa acerca del estado del sistema.