



**2º de Grado en Ingeniería Informática**  
**Escuela Politécnica Superior**  
**Universidad de Córdoba**  
**Departamento de Informática y Análisis Numérico**



**Sistemas operativos, apuntes de la asignatura.**

Profesor: Juan Carlos Fernández Caballero

email: [jfcaballero@uco.es](mailto:jfcaballero@uco.es)

## **Tema 1- Introducción a los Sistemas Operativos**

# 1. Índice de contenidos

1	¿Qué es un sistema operativo?	1
1.1	El sistema operativo como gestor-asignador de recursos	1
1.2	El sistema operativo como interfaz simple intermediaria	3
2	Diseño modular en niveles	4
3	Transformación de un programa en instrucciones máquina	5
3.1	Ejemplo de paso de lenguaje ensamblador a lenguaje máquina	7
4	Elementos básicos y organización de un computador	8
4.1	El procesador y sus registros	9
4.2	Interacción de la CPU con la memoria principal	11
4.3	Estructura de Entrada/Salida (E/S): drivers y controladores	12
5	Búsqueda y ejecución de instrucciones	13
5.1	Ejemplo de ejecución de instrucciones	14
6	Interrupciones y manejador de interrupciones	16
6.1	Adición de la fase de interrupción	18
6.2	Interrupciones múltiples	21
7	Sistemas multiprocesador	23
7.1	Sistemas multiprocesador simétricos	25
8	Multiprogramación	26
9	Multiprocesamiento	27
9.1	HyperThreading	28
10	Modo dual (usuario y <i>kernel</i> )	29
11	Llamadas nativas al sistema y paso de parámetros	31
11.1	Pasos generales en la invocación de una llamada nativa al sistema mediante funciones de tipo TRAP	35
11.2	Ejemplo de llamadas nativas al sistema en sistema GNU/Linux clásico	37
11.3	Localizar cuáles son los números (IDs) de llamadas nativas al sistema en un SO GNU/Linux actual	40
12	Intérprete de comandos	42
13	Un poco de historia	43

## 1 ¿Qué es un sistema operativo?

Como definición básica utilizada durante muchos años, un **Sistema Operativo** es **conjunto de programas en ejecución (procesos) que administran el hardware de una computadora**, siendo todo lo demás programas del sistema y programas de aplicación.

Un **proceso** es un **programa** cargado en **memoria principal**, y el Sistema Operativo debe estar cargado en memoria principal y disponer de CPU para ejecutarse como cualquier otro proceso.

Recuerde que **hay dos tipos de programas**:

- Los *programas de sistema*, que se encargan de controlar las operaciones propias de la computadora, por ejemplo, el Sistema Operativo, los demonios y procesos de servicios.
- Los *programas de aplicación*, que resuelven problemas específicos a los usuarios (procesador de textos, navegador, etc).

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**:

- **Gestionar y planificar el uso, por parte de los distintos procesos activos, de los recursos hardware y servicios del sistema** (espacio de memoria principal/secundaria/virtual, dispositivos de E/S, procesadores, cola de impresión, bluetooth, etc), sin que el usuario tenga que preocuparse de los mismos.
- Proporcionar una **interfaz intermediaria entre la máquina o hardware y el usuario**, con la que es factible comunicarse, ocultando la complejidad del hardware.

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

Al enfrentarse a numerosas y posiblemente **conflictivas solicitudes de recursos**, el sistema operativo debe decidir cómo asignar éstos a procesos y usuarios específicos, de modo que la computadora pueda operar de forma eficiente y equitativa. Cualquier **planificación y asignación de recursos** por parte del sistema operativo **debe tener en cuenta tres factores**:

- **Equitatividad**: Ante **procesos similares**, es deseable que todos los procesos que compiten por un determinado recurso se les conceda un acceso equitativo al mismo (**evita inanición**).
- **Respuesta diferencial**: Ante **procesos diferentes** y con **distinta prioridad**, el sistema operativo puede necesitar **discriminar** a la hora de asignar recursos entre ellos, y debe tomar las decisiones **de forma dinámica**.

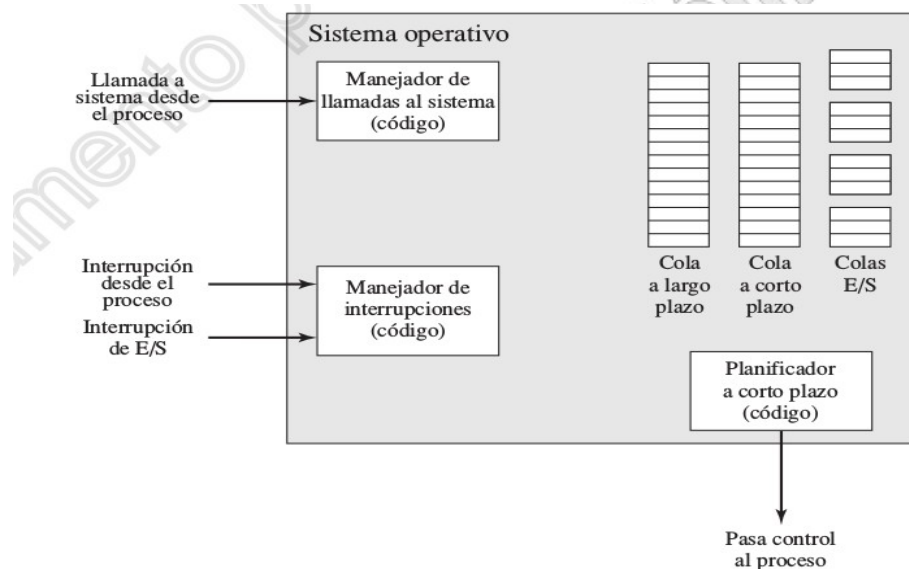
Por ejemplo, si hay un proceso que está ocupando el uso de un dispositivo de E/S y aparece otro proceso de mayor importancia y prioridad que necesita el mismo recurso, el sistema operativo puede intentar planificar (meter en CPU) el nuevo proceso y sacar momentaneamente al que está usando el dispositivo de E/S.

- **Eficiencia**: El sistema operativo debe intentar: 1) **maximizar la productividad** (número de procesos terminados por unidad de tiempo), 2) **minimizar el tiempo de respuesta** (unidades de tiempo que tarda un proceso en ser atendido). Estos **criterios entran en conflicto**, hacer que cada proceso tenga un tiempo de respuesta corto puede hacer que el número de procesos terminados por unidad de tiempo disminuya. Medir la actividad del sistema es importante

para ser capaz de monitorizar el rendimiento y realizar los ajustes correspondientes.

De manera genérica la Figura 2.11 sugiere los principales **elementos del sistema operativo** para **llevar a cabo los tres factores de equitatividad, respuesta diferencial y eficiencia**:

- **Planificador**: Hace referencia a la planificación de procesos y la asignación de recursos en un entorno de multiprogramación (intercambio de procesos en CPU con un solo núcleo).
- **Manejador de llamadas nativas al sistema**: La gestión adecuada de las **llamadas nativas al sistema** (funciones propias del sistema operativo llamadas a partir de las APIs a nivel de usuario, las cuales actúan como envoltorios).
- **Manejador de interrupciones**: Las **interrupciones** provocadas por **dispositivos de E/S** (por ejemplo se dispone de un dato de disco duro que espera un proceso) se deben gestionar de la mejor manera posible, invirtiendo en ellas el menor tiempo o ciclos de reloj posible. También existen interrupciones por **temporización** (tareas regulares del sistema operativo, tiempo de ejecución en CPU asignado a un proceso finalizado). Y también existen interrupciones provocadas por la ejecución de alguna **operación errónea** por parte de un proceso, como una división por cero.



**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 en cuanto a planificación**, 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** 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**, pero que **no están cargados por completos en memoria principal** hasta que no pasan a la cola de corto plazo. El sistema operativo añade trabajos al sistema transfiriendo un trabajo desde la cola a largo plazo hasta la cola a corto plazo. Por tanto, el sistema operativo debe

estar seguro de que no sobrecarga la memoria principal o el tiempo de procesador admitiendo demasiados procesos en el sistema.

Hay también **colas de E/S** por cada dispositivo de E/S, ya que más de un proceso puede solicitar el uso del mismo dispositivo. Contienen información sobre los procesos que están haciendo o quieren hacer uso de un determinado dispositivo, de forma que todos los procesos que esperan utilizar dicho dispositivo se encuentran alineados en su correspondiente cola. 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, pero a nivel del uso del procesador. 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 interfaz simple intermediaria

La **utilización directa del hardware es una tarea difícil para un programador**, 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 ordenes, 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 ordenes 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 que es más fácil de programar que el hardware subyacente**.

Dicho esto, 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. De esta manera, un programador o un usuario ve una abstracción del hardware que entiende ordenes de nivel superior a partir de esa máquina virtual o **interfaz intermedia**. Por ejemplo, **llamadas a funciones de lectura o escritura en un disco duro a partir del conjunto de funciones o API (Application Programming Interfaces) de un lenguaje de programación de alto nivel**.

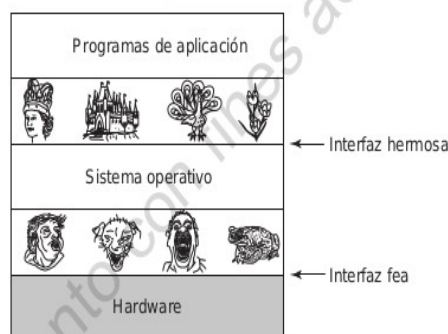


Figura 1-2. Los sistemas operativos ocultan el hardware feo con abstracciones hermosas.

**El sistema operativo, como máquina virtual, debe proporcionar los siguientes servicios:**

- **Ejecución de programas:** Para poder ejecutar un programa se tienen 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 (DRIVER). 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 y comunicación con los periféricos. Todo ello es transparente para el usuario final.
- **Manipulación y control del sistema de archivos:** Además de comunicarse con el controlador del periférico donde está el sistema de archivos, el sistema operativo debe conocer la propia **estructura y formato de almacenamiento del periférico** (ntfs, ext4, fat32, big endian, little endian, etc), y proporcionar los mecanismos adecuados para su control y protección.
- **Detección de errores:** Hay una gran cantidad de errores que pueden ocurrir, tanto del hardware como del software. 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 (permisos)**. 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 (monitoreo):** 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 (este tipo de estadísticas pueden ser utilizadas incluso por el **planificador** del sistema).

## 2 Diseño modular en niveles

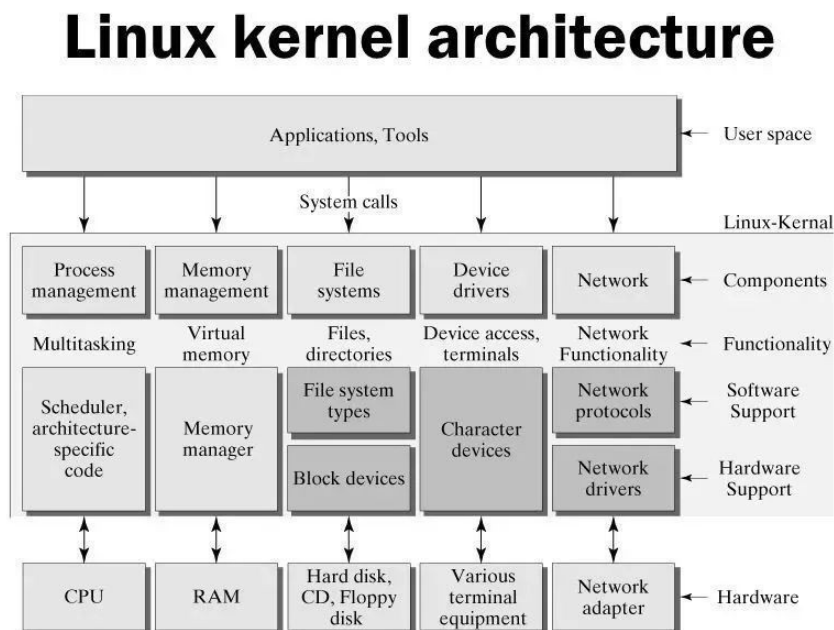
Los sistemas operativos puede dividirse en partes más pequeñas y más adecuadas que lo que permitían sistemas más antiguos como por ejemplo MS-DOS o UNIX.

Con un **método de diseño modular** las características y la funcionalidad globales del sistema se separan en componentes, de esta manera:

1. Un sistema modular ofrece **mayor libertad de cambio y de ampliación del sistema**.
2. Un sistema modular ofrece **menor esfuerzo de mantenimiento**.
3. Un sistema modular ofrece **mayor facilidad de corrección de errores y depuración**.

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. Es decir, un nivel no necesita saber cómo se implementan dichas rutinas, sólo necesita saber qué hacen, de manera que cada nivel oculta a los niveles superiores la existencia de determinadas estructuras de datos, operaciones o hardware.

La siguiente figura muestra un esquema del diseño modular de un sistema GNU/Linux, donde se pueden visualizar diferentes módulos que gestionan a diferentes recursos del sistema.

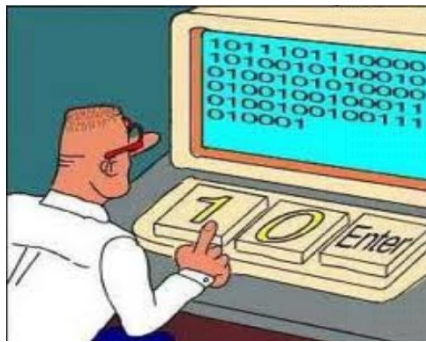


### 3 Transformación de un programa en instrucciones máquina

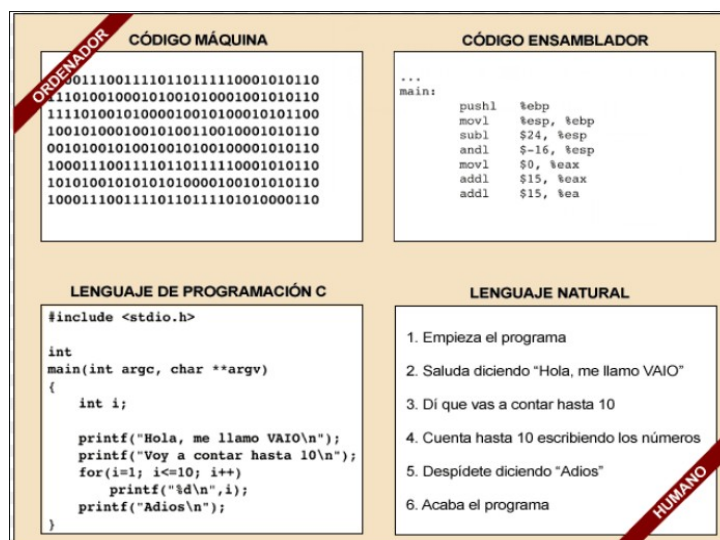
Las máquinas interpretan más fácilmente las señales **on** y **off**, lo que equivale a interpretar la presencia o la ausencia de voltaje. Así, el alfabeto de las máquinas está constituido por dos símbolos que son representados por **1** y **0** respectivamente. A cada una de ellos se le conoce como 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 que implican ordenes o acciones. 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:





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**.



Para que los computadores puedan procesar los diferentes tipos de lenguajes de programación se desarrollaron otros programas que traducen los programas escritos a alto nivel a un lenguaje de más bajo nivel, estamos hablando de los **compiladores**.

En la figura anterior se muestra un ejemplo de un programa escrito en un lenguaje de alto nivel como C, que es traducido por medio del compilador a un programa en **lenguaje ensamblador** (a cada línea se le denomina "mnemónico"), que estaría más bajo nivel. Lea la figura de derecha a izquierda, empezando por la parte inferior.

El lenguaje ensamblador todavía no es un lenguaje entendible por la máquina. Por otra parte, **ensamblador** es también un tipo de programa informático, parecido a un compilador, que se encarga de traducir un fichero fuente escrito en **lenguaje ensamblador** a un **fichero objeto que contiene código máquina (código binario)** y que de nuevo estaría a un nivel inferior.

El **código máquina** está formado por un **conjunto** de lo que se denominan **instrucciones** (cargadas en memoria principal en forma de unos y cero), que determinan una serie de acciones que debe realizar la máquina. Por ejemplo, el programador escribe en lenguaje ensamblador "**add A, B**" 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.

### 3.1 Ejemplo de paso de lenguaje ensamblador a lenguaje máquina

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` y `01100001`.

- El primer byte contiene empaquetado la instrucción **MOV** y el código del registro (AL) hacia donde se va a mover el dato. (TIPO DE OPERACIÓN)
- En el segundo byte se especifica el número `61h`, escrito en binario como `01100001`, que se asignará al registro AL (DATO)

```
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** (e incluso de manera dinámica en tiempo de ejecución) para así poder proporcionar servicios y gestionar recursos para los procesos. 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**), 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.

## 4 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 controla el funcionamiento del computador y realiza sus funciones de procesamiento de datos, ejecutando instrucciones almacenadas en memoria principal. La CPU contiene:
  - Una unidad aritmético lógica (ALU).
  - Un registro de instrucciones (IR).
  - Un contador de programa (PC).
  - Otra serie de registros auxiliares (RDIM, RDAM, RDI E/S, RDA E/S).
- 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.

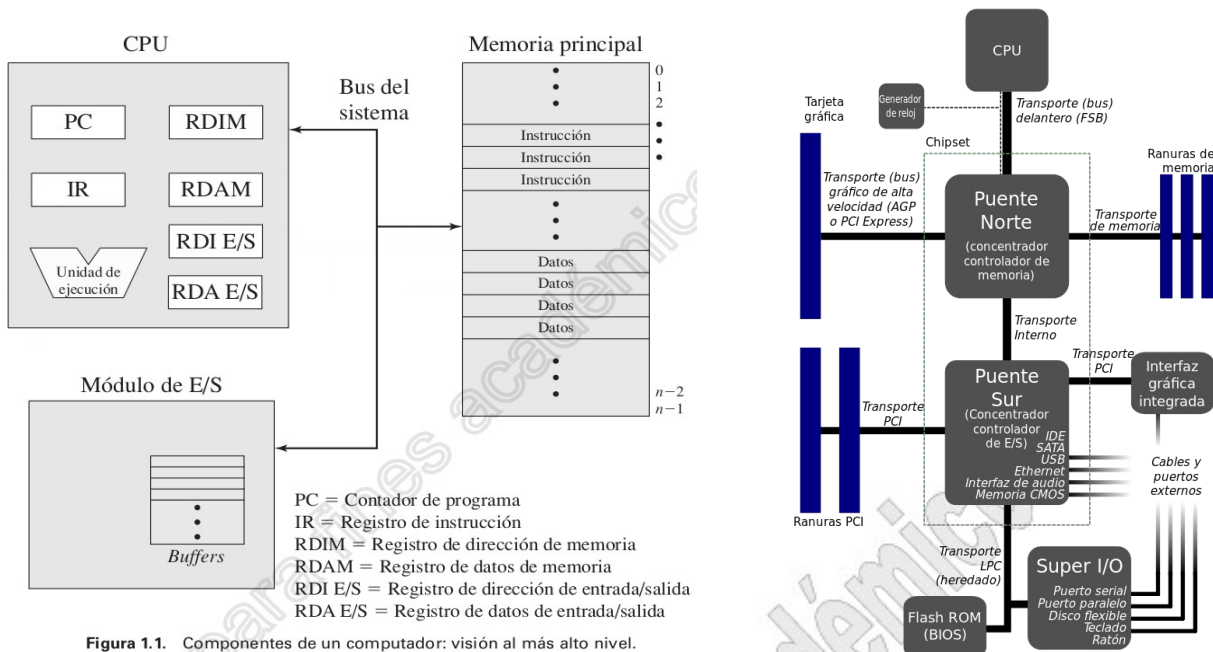
Un módulo de memoria consta de un conjunto de posiciones definidas mediante direcciones numeradas secuencialmente. Cada posición contiene un patrón de bits que se puede interpretar como una instrucción o como datos.

- **Módulos o controladoras de entrada y salida (E/S)**. Es una parte hardware que transfiere los datos entre el computador y su entorno externo y viceversa. El entorno externo está formado por diversos dispositivos, incluyendo dispositivos de memoria secundaria (discos duros, CD-ROMS, *pendrives*, etc), equipos de comunicaciones y terminales.

Un módulo o controladora de E/S **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.

Todos estos componentes se interconectan de manera que se pueda lograr la función principal del computador, que es ejecutar programas. Para esa interconexión 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:



La CPU y las controladoras de dispositivos pueden funcionar de forma **concurrente** (a la misma vez), **compitiendo por la memoria principal**. Para asegurar el acceso de forma ordenada a la memoria principal, se proporciona una **controladora de memoria** (no se muestra en la figura 1.1 izquierda) cuya función es sincronizar el acceso a la misma.

Las placas base modernas suelen incluir dos **chips o circuitos integrados para la comunicación entre componentes de la placa** (además de los buses), denominados **punto norte** y **punto sur**, y suelen ser los circuitos integrados más grandes después de la CPU y la GPU (Unidad de Procesamiento de Gráficos). La controladora de memoria puede ser un chip independiente (normalmente en el punto norte de la placa) o incluso estar integrado en la CPU (con el objetivo de reducir la latencia y el consumo). Las últimas placas base carecen de punto norte, ya que los procesadores de última generación tienden a llevarlo integrado.

#### 4.1 El procesador y sus registros

Una de las funciones del procesador es el **intercambio de datos con la memoria y los dispositivos de E/S** a través del módulo de E/S. Para este fin se utilizan un conjunto de registros de las CPU que proporcionan un tipo de memoria que es más rápida y de menor capacidad que la memoria principal:

- Un **registro de dirección de memoria (RDIM o MAR)**, que puede almacenar temporalmente una *dirección de memoria principal en la que escribir o de la que leer*.
- Un **registro de datos de memoria (RDAM o MDR)**, que puede almacenar temporalmente *datos que se reciben de la memoria principal o que se pueden escribir en ella*.
- Un **registro de dirección de E/S (RDIE/S)**, que puede almacenar temporalmente una

*dirección de memoria de los buffers del módulo de E/S en la que escribir o de la que leer.*

- Un **registro de datos de E/S (RDAE/S)**, que puede almacenar temporalmente *datos que se reciben de los buffers del módulo de E/S o que se pueden escribir en ellos.*

Además de los registros mencionados anteriormente, los siguientes son esenciales para la ejecución de instrucciones:

- **Registro de instrucción (Instruction Register, IR).** Contiene la última instrucción leída y que hay que ejecutar, es decir, el contenido de la última dirección de memoria cargada y esa dirección de memoria la indica otro registro que se verá a continuación llamado PC. En el registro IR son analizados el código de operación y los campos de operando.
- **Contador de programa (Program Counter, PC).** Contiene una dirección de memoria a cargar, concretamente la de la próxima instrucción que se cargará desde la memoria principal. Típicamente, la CPU actualiza el PC después de cada captación de instrucción en el registro IR, de manera que apuntará a la siguiente instrucción a cargar y 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.
- 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 aritmético**, por lo que además de almacenarse el resultado en sí mismo en un registro o en la memoria, se fija también en un registro para cada caso un código de condición en concordancia con el resultado de la ejecución de la instrucción. Posteriormente, se puede comprobar el código de condición como parte, por ejemplo, de una operación de salto condicional o para realizar una acción determinada ante esa condición cumplida.

- **Otro tipo de registros:** Además de los registros fundamentales anteriores, los procesadores disponen de otra serie de registros tanto para tareas de **almacenamiento temporal**, como para el **direccionamiento, control y condiciones** de las operaciones que se realizan, como por ejemplo:
  - **Acumulador (AC):** Para el almacenamiento de resultados de cálculos matemáticos realizados por la ALU de la CPU.
  - **Registros de propósito general (AX, BX, CX, DX):** Sirven para almacenar temporalmente los bits de datos o instrucciones.
  - **Punteros de pila:** Para el proceso que esté en ejecución habrá un registro dedicado a señalar la cima de la pila y otro registro para la base (se verá más adelante).

Los registros anteriores se pueden clasificar en función de si **son visibles o no al usuario**:

- **Registros visibles para el usuario (se acceden en modo usuario).** Se permite su acceso por parte del programador mediante lenguaje máquina o en ensamblador. 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. Dependiendo de la arquitectura del procesador, estos registros pueden cambiar. Los tipos de registros que están normalmente disponibles son algunos registros de datos (por ejemplo el AC) o de dirección (contienen direcciones de memoria principal de datos e instrucciones) .
- **Registros de control y estado no visibles para el usuario (reservados para la ejecución de instrucciones en modo núcleo).** Son 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), pero nunca se accede a ellos directamente por parte del programador o programas de usuario.

## 4.2 Interacción de la CPU con la memoria principal

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, donde cada palabra tiene su propia dirección.

La **interacción con la memoria principal** se consigue a través de:

- 1) **Una secuencia de carga (*load*), desde memoria a CPU:** La instrucción *load* mueve una palabra desde la memoria principal a un registro interno de la CPU (IR),
- 2) **Una secuencia de almacenamiento (*store*) de instrucciones, desde CPU a memoria:** La instrucción *store* mueve el contenido de un registro de la CPU a la memoria principal.

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 y el disco de estado sólido (**SSD**). Estos disco 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 los CD-ROM, pendrive, tarjetas SD, etc. Las principales diferencias entre los distintos sistemas de almacenamiento están relacionadas con el coste, el tamaño, la volatilidad y su velocidad, llegando al caso más extremo con la memoria caché de la CPU.

### 4.3 Estructura de Entrada/Salida (E/S): drivers y controladores

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**, que son **muy lentos comparados con la velocidad de la CPU**. 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, su búfer local y la CPU.

En las siguientes figuras se pueden ver los interfaces de conexión entre los módulos o controladores de E/S y los dispositivos periféricos. Los módulos de E/S en si mismos se encuentran en los chips del puente norte y puente sur.

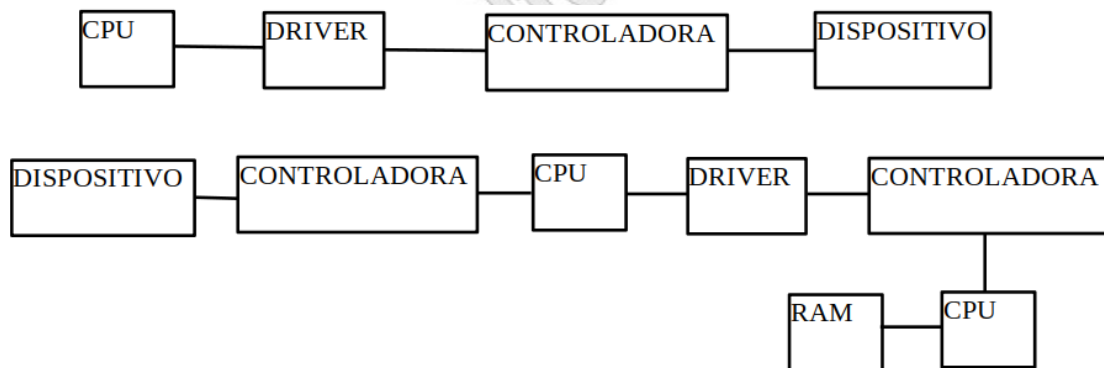


Normalmente, los sistemas operativos tienen al menos un **controlador/a de dispositivo software (llamado driver)** para cada **controlador/a de dispositivo hardware (llamado controller)**. Es muy importante **no confundir** la **controlador/a de dispositivo hardware** con la **controlador/a de dispositivo 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 que la controladora hardware y el dispositivo se entiendan, y presenta por tanto al sistema operativo una interfaz uniforme mediante la cual comunicarse con el dispositivo.



De forma resumida y genérica, el **driver** y la **controladora de dispositivos funcionan así**:

1. Al iniciarse una operación de E/S, la CPU carga el *driver*.
2. El *driver* a su vez carga los registros apropiados con las instrucciones adecuadas en la controladora hardware, en función de la operación a realizar que le haya indicado la CPU, por tanto ha adaptado la operación al tipo de dispositivo físico concreto con el que se va a trabajar.
3. Posteriormente la controladora hardware examina el contenido de estos registros para determinar qué acción se le ha comunicado, por ejemplo leer un fichero desde un tipo de lector de CD-ROM especial. La controladora inicia entonces la operación con el dispositivo, en este ejemplo concreto indicar al CD-ROM qué tiene que leer y transferir de datos leídos desde el dispositivo a su búfer local.
4. Una vez completada la función o tarea a realizar por parte del dispositivo, éste avisa a la controladora de su finalización.
5. La controladora a su vez informa a la CPU mediante un envío de señal de interrupción de que ha terminado la operación que se le encomendó.
6. La CPU carga el *driver*.
7. El *driver* adapta los datos de la controladora y los devuelve entonces hacia la CPU.
8. A continuación la CPU los pasa a memoria RAM. Para otras operaciones, el *driver* devuelve solo información de estado, como por ejemplo OK o Error.



## 5 Búsqueda y 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 cada una instrucción consta de dos pasos: el procesador lee instrucciones de la memoria, una cada vez, y la ejecuta.

Desde el punto de vista de los registros del procesador, 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 (a no ser que se carguen instrucciones de salto).

Considere, por ejemplo, un computador simplificado en el que cada instrucción ocupa una palabra de memoria de 16 bits. Suponga que el PC 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, aunque esta secuencia se podría *alterar con instrucciones de tipo salto*.

Una instrucción de salto especifica 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 será la de 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.

Pues bien, se denomina **ciclo de instrucción** al procesamiento requerido por una única instrucción (conlleva varios ciclos de reloj) y formado por dos pasos (ver Figura 1.2) llamados **fase de búsqueda y fase de ejecución**.

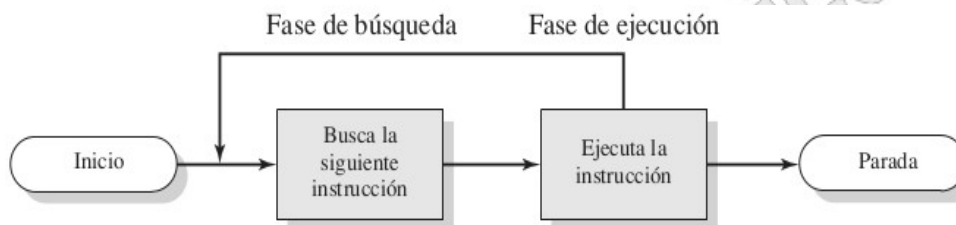


Figura 1.2. Ciclo de instrucción básico.

La ejecución de ciclos de instrucción en un **sistema** se detiene sólo si se **apaga la máquina** o se produce algún tipo de **error irrecuperable**.

Si se está ejecutando un **proceso** (puede contener cientos o miles de instrucciones), la ejecución iterativa de ciclos de instrucción se termina si se ejecuta una instrucción que haga **salir al proceso de la CPU**, por ejemplo por la espera de un evento, interrupción por temporización, finalización natural del proceso, división por cero, etc.

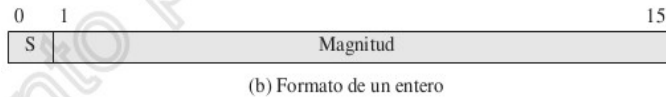
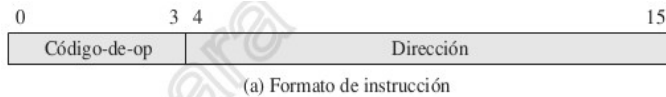
## 5.1 Ejemplo de ejecución de instrucciones

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

- El procesador contiene un único **registro de datos**, llamado el acumulador (**AC**), más en registro **IR** y **PC**.
- Las **instrucciones** y también 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).
- Con los **12 bits restantes** del formato de la instrucción se pueden **direccionar** directamente hasta  $2^{12} = 4.096$  (**4K**) **palabras de memoria** (en RAM puede haber instrucciones o datos),

denotadas por tres dígitos hexadecimales.

La Figura 1.4 ilustra una ejecución parcial de las instrucciones (en hexadecimal) de un programa, mostrando las partes relevantes de la memoria y de los registros del procesador.



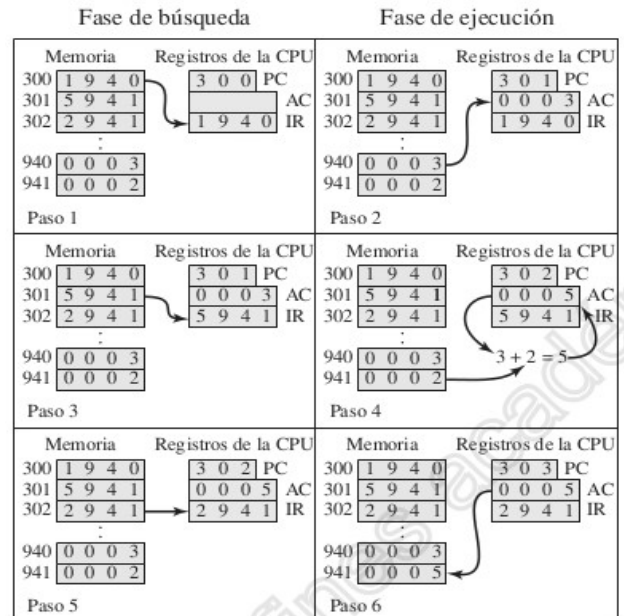
Contador de programa (PC) = Dirección de la instrucción  
 Registro de instrucción (IR) = Instrucción que se está ejecutando  
 Acumulador (AC) = Almacenamiento temporal

(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.



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

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, **941 ← 940 + 941**.

Para este cometido 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:

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.  
 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.
2. Se carga en AC el contenido de la dirección de memoria 940, que es 0003 en hexadecimal.
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.

## 6 Interrupciones y manejador de interrupciones

Mientras el procesador esta 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 previamente 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 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 también 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. En relación a esto también hay algoritmos de planificación que **introducen en CPU a un proceso si este es más prioritario**, por lo que el proceso que estuviera actualmente ejecutando se vería interrumpido para meter a ese otro proceso más prioritario.

- **Interrupciones de proceso (excepciones):** 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, problemas graves en el sistema, operación que ya no es posible realizar, etc.

En realidad **las excepciones no son interrupciones como tales**, las cuales suelen usarse como recursos para atender necesidades de los sistemas operativos y de los procesos de usuarios, sino que una excepción es una **interrupción provocada por una situación de error detectada por la CPU mientras ejecutaba una instrucción** y que requiere tratamiento por parte del SO, causando en general la terminación del proceso en curso que la causó.

Hay excepciones que si se producen en programas a nivel de usuario, como la división por cero, no abrir un fichero correctamente, acceso a un array fuera de rango, etc, pueden controlarse de forma que el programa que la causa pueda continuar con su ejecución o provoque una salida ordenada y adecuada.

De manera general, cuando se interrumpe a la CPU (lo detecta a través de un bit de la palabra de estado) 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, se trata la interrupción (por ejemplo obtener una información solicitada anteriormente a un dispositivo de E/S), y cuando se ha terminado se le devuelve el control a la CPU, la cual reanuda la operación o proceso que estuviera ejecutando antes de la interrupción (o cualquier otro que el sistema operativo considere necesario) o se finaliza si es una excepción.

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**.

- Para ello, se dispone de una **tabla de punteros a las diferentes rutinas de interrupción** (se le suele llamar **vector de interrupciones**). Concretamente, un vector de interrupciones es una tabla o matriz ubicada en memoria principal, donde cada elemento contiene la **dirección de memoria** de las **diferentes rutinas ISR** que se pueden ejecutar.

El **vector de interrupciones se indexa** mediante un **número de interrupción unívoco**, que se proporciona en la propia solicitud de interrupción, por medio de una controladora hardware, 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 GNU/Linux 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 y si se decide así por la política de planificación en uso, la dirección de retorno guardada se carga en el contador de programa (PC), 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.

Note que cuando estamos hablando de interrupción el “sentido” va hacia el procesador, es decir, el procesador esta 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 resultará en una o varias llamadas nativas al sistema (se estudiará posteriormente), pero eso **NO ES UNA INTERRUPCIÓN**, aunque como veremos, implica que el proceso que llama tenga que salvarse para pasar a ejecutarse la llamada nativa.





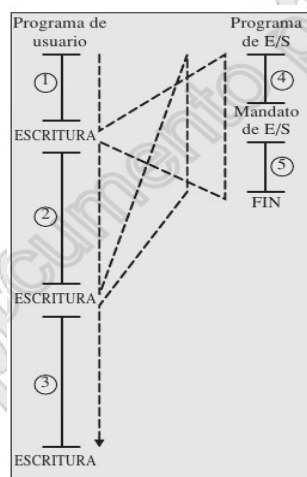
## 6.1 Adición de la fase de interrupción

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

Para dar un ejemplo concreto de la lentitud de un dispositivo de E/S, considere un computador personal que opere a 1GHz, lo que le permitiría ejecutar aproximadamente  $10^9$  instrucciones por segundo. Un disco duro básico es 4 millones de veces más lento que el procesador.

Suponga un proceso en el que el procesador está transfiriendo datos a una impresora. **Sin el concepto de interrupción ni módulos de E/S**, 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 informe de ello). 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 el tener que esperar a que el *buffer* de la impresora esté vacío para volver a enviarle información y llenarlo, o simplemente esperar a que se indique si ese vaciado ha tenido algún problema.

Continuando con el ejemplo anterior de la impresora, suponga que el proceso realiza una serie de peticiones de impresión (indicado con la palabra ESCRITURA) intercaladas con el procesamiento del resto del código del programa (ver Figura).



(a) Sin interrupciones

El **circulo 4** consiste en preparar la operación de E/S, es decir, en invocar las rutinas necesarias y ejecutar las instrucciones pertinentes para comprobar si la impresora está disponible y enviarle la información a imprimir a sus *buffers*.

El **circulo 5** es la operación en si, es decir, que la impresora vacíe su *buffer* para imprimir los datos e indique al sistema operativo que todo ha ido correcto o que ha habido algún error.

Debido a que la operación de vaciado e informe 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, “desperdiciándose” el uso de la CPU hasta que se vacíe el *buffer* de la impresora y se informe de ello. Esto por tanto sería una **manera ineficiente de comunicación con los periféricos** y se dejaría



ociosa a la CPU durante muchos ciclos de reloj.

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

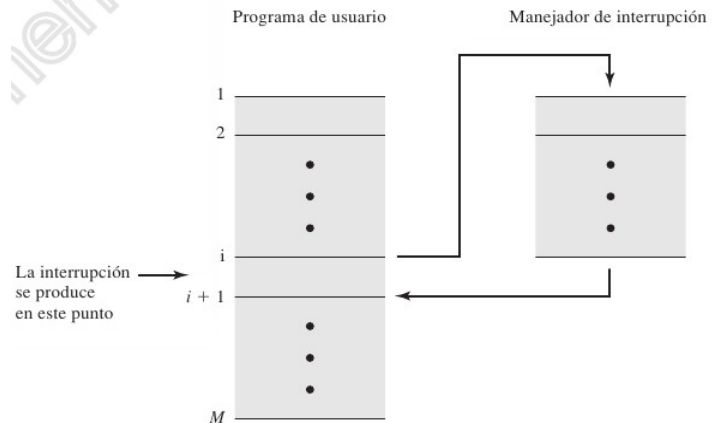
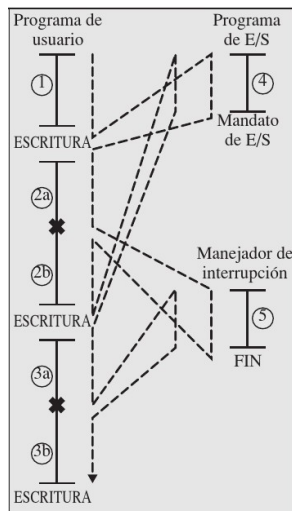


Figura 1.6. Transferencia de control mediante interrupciones.

En este ejemplo y siguiendo esta última idea, el **circulo 4** corresponde con el envío de la CPU al **módulo de E/S** de las instrucciones pertinentes con el envío de la información que necesita imprimirse.

Acto seguido la CPU continuará con la siguiente línea de código del programa que se está ejecutando, y es el módulo de E/S el que se pone en contacto con la impresora y le transfiere la información que le ha dado la CPU.

Cuando la impresora haya terminado de vaciar su buffer se lo indica al módulo de E/S y este a su vez manda una petición de interrupción al sistema operativo a través de la CPU, 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 (a través del controlador de E/S) hacia el procesador.

El **circulo 5** se corresponde ahora con el tratamiento de la interrupción, es decir, con la **ejecución de la rutina ISR**. En este caso podría ser indicar por ejemplo un OK de que el *buffer* de la impresora está vacío y ya puede imprimir más cosas.

Cuando se completa el tratamiento de la interrupción de nuevo **se reanuda la ejecución del proceso por donde iba**, aunque dependiendo de la política de planificación del sistema se podría continuar con otro proceso, o incluso con otra interrupción como se verá más adelante.

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.

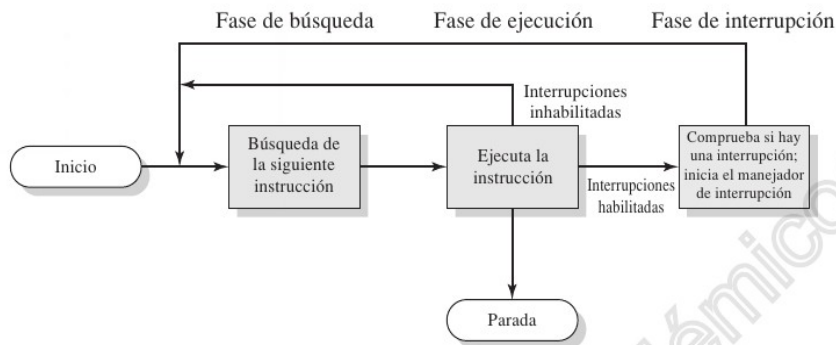


Figura 1.7. Ciclo de instrucción con interrupciones.

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**. 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, o de otro proceso que se crea conveniente según la política de planificación.

En el caso de que no haya interrupciones pendientes después de la fase de comprobación, 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.

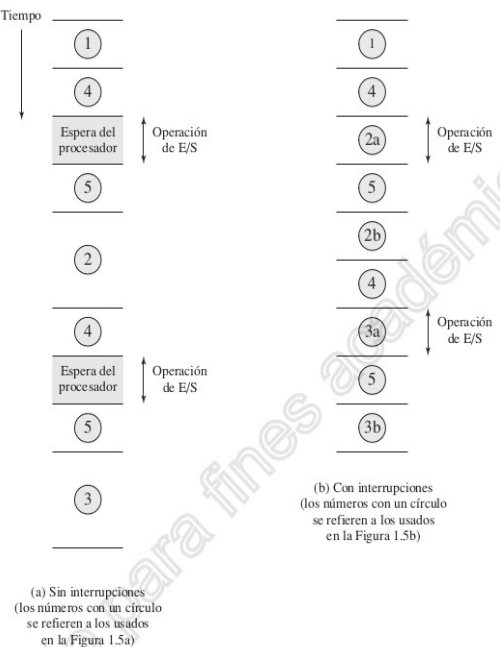


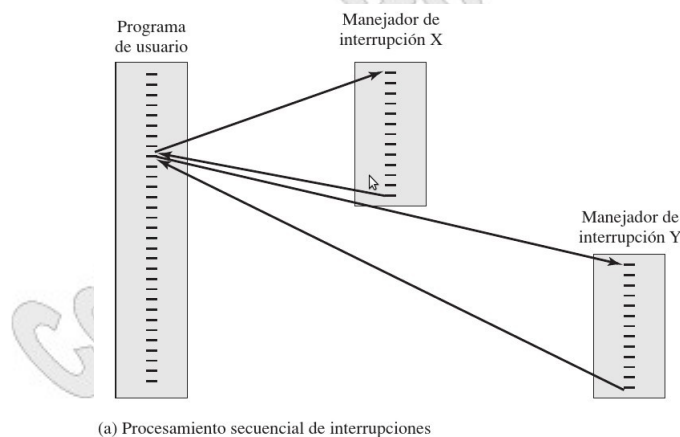
Figura 1.8. Temporización del programa: espera breve de E/S.

## 6.2 Interrupciones múltiples

El estudio realizado hasta el momento ha tratado solamente el caso de que se produzca una única interrupción. Suponga, sin embargo, que se producen **múltiples interrupciones**, es decir, mientras se está tratando por *ISR* una interrupción se produce otra. Por ejemplo, un sistema 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 atenderá después de que se rehabiliten 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 rehabilitan las interrupciones antes de reanudar el programa de usuario, y el procesador comprueba si se han producido interrupciones adicionales para atenderlas. Esta estrategia es válida y sencilla, puesto que las interrupciones se manejan en estricto orden secuencial (Figura 1.12a). Note que de alguna manera debe existir alguna estructura de datos y algún mecanismo hardware para controlar ese orden.



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, 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 (prioridad 2)**; se almacena la información del contexto del proceso actual y la ejecución continúa en la ISR de la impresora.
3. Mientras todavía se está ejecutando esta rutina, en  $t = 15$  se produce una **interrupción del equipo de comunicaciones (prioridad 5)**. 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 la ISR del equipo de comunicaciones se produce una **interrupción del disco en  $t = 20$  (prioridad 4)**. 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 ( $t = 25$ ), 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 devuelve finalmente el control al programa de usuario (previa restauración de su contexto).

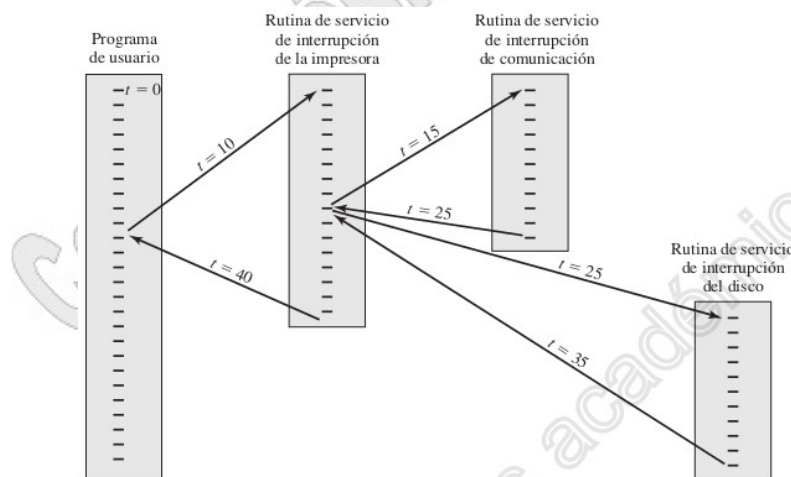


Figura 1.13. Ejemplo de secuencia de tiempo con múltiples interrupciones.

Como **desventaja**, 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.

## 7 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.

En tales sistemas los procesadores 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.

Además de esa CPU principal de propósito general, casi todos los sistemas disponen también de otros **“microprocesadores” de propósito especial**, los cuales pueden venir en forma de controladores 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 el, como por ejemplo, transmitir datos entre la memoria principal y un dispositivo de E/S mientras que el procesador principal realiza otras tareas.

Tenga en cuenta que estos procesadores de propósito especial pueden ejecutar un conjunto limitado de instrucciones, pero **no ejecutan programas de usuario**, esto último solo lo hace el procesador principal. **Es el sistema operativo quien les envía información sobre su siguiente tarea y monitoriza su estado**. Por ejemplo, un microprocesador incluido en una controladora de disco (controladora de E/S) recibe una secuencia de solicitudes procedentes de la CPU principal e implementa su propia cola de disco y su algoritmo de programación de tareas, es decir, ante muchos procesos que intentan acceder a disco, la controladora podría tener una política de planificación y control de acceso. Este método libera a la CPU principal del trabajo adicional de planificar las tareas de disco y la libera de tiempos ociosos.

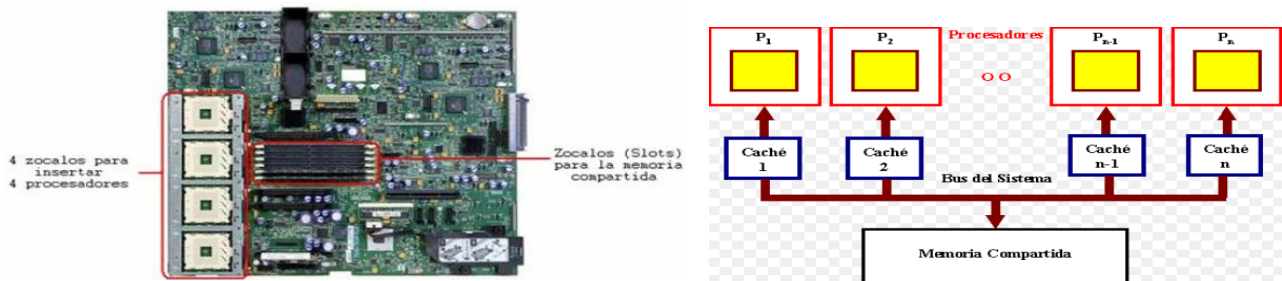
La presencia de microprocesadores de propósito especial resulta bastante común, pero no convierte a un sistema en un sistema multiprocesador. Si sólo hubiera una CPU de propósito general con un solo núcleo, entonces el sistema sería de un solo procesador.

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, de forma que la estrecha proximidad de estos nanotransistores 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 más pequeños y menos rápidos. 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 núcleos es más lento que un chip con un solo núcleo más grande.

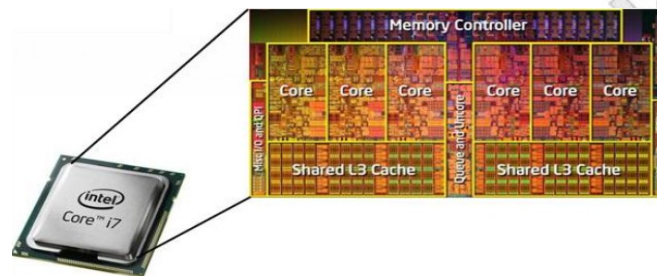
Hay que decir que esta mejora tecnológica tiene un problema, y es que **el sistema operativo debe tener mecanismos que permitan trabajar con varios núcleos y/o varios procesadores** para que compartan RAM, periféricos etc, lo cual complica enormemente su implementación y diseño.



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 solo 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 realicen más trabajos independientes en menos tiempo, y así es, con lo cual ya es una gran ventaja. Sin embargo, la mejora en rendimiento 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.



## 7.1 Sistemas multiprocesador simétricos

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**:

- 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 (lo gestiona el núcleo del Sistema Operativo). El sistema operativo se encarga de planificar los hilos o procesos en procesadores individuales, y de la sincronización entre los procesadores.

SMP tiene más ventajas 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 no 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 su interacción.

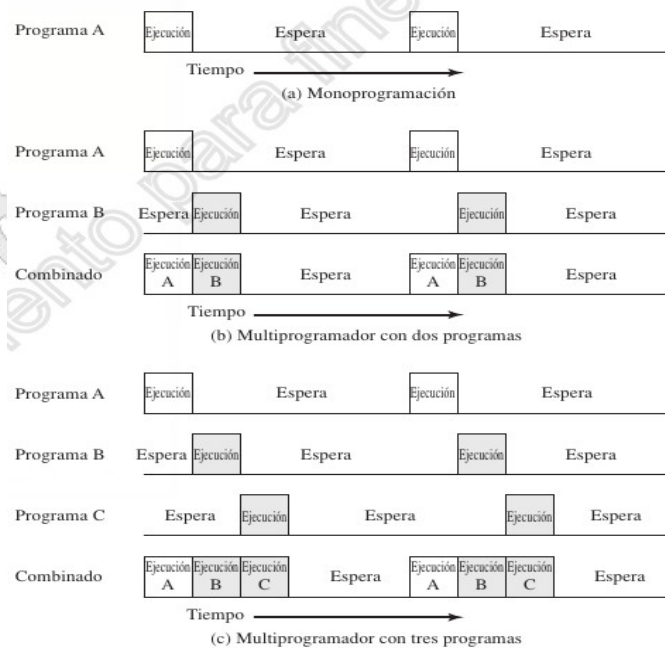
## 8 Multiprogramación

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 simultanea, 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, como 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.

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 y que realizan operaciones de E/S**. 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.



**Figura 2.5.** Ejemplo de multiprogramación.



## 9.1 HyperThreading

*HyperThreading* es una marca registrada de la empresa Intel para nombrar su implementación de la tecnología *Multithreading Simultáneo*, también conocida como SMT. *HyperThreading* permite ejecutar dos hilos dentro de un único núcleo del procesador. Se estudiará más adelante la diferencia entre un proceso y un hilo. Los procesos pueden estar formados por un solo hilo de ejecución o por varios hilos.

Esta tecnología hardware consiste en **simular en cada núcleo físico del procesador dos procesadores lógicos**, dividiendo los hilos entre ambos y por tanto mejorando la velocidad de procesamiento. El resultado es una **mejoría en el rendimiento del procesador**, y esto es posible por la duplicación en la CPU de los registros de propósito general, registros de control (PC, IR), algunos registros de estado y otros componentes de arquitectura, pero comparten algunos niveles de caché, buses y ALU. En un sistema sin *HyperThreading* encontramos ciclos de instrucción donde la ALU se encuentra ociosa esperando recibir datos, pero con *HyperThreading* se podría utilizar por otro hilo que necesite esa ALU.

El sistema operativo ha de estar preparado para utilizar esta tecnología. Actualmente los sistemas operativos modernos GNU/Linux, Mac OS y Windows lo implementan en sus núcleos al soportar modelos multihilo o SMT.

*HyperThreading* conlleva una mejora en la velocidad de las aplicaciones, que según Intel es aproximadamente de un 30 por ciento más comparado con un procesador que no implemente *HyperThreading* en sus núcleos.

Respondamos entonces a estas preguntas:

- **¿Existe paralelismo real en un núcleo con tecnología Hyperthreading (supongamos una CPU con un solo núcleo)?**

Existe paralelismo real solo cuando dentro de un núcleo, los hilos de ejecución que haya no tengan que utilizar a la vez aquellas partes hardware NO replicadas, ya que en un núcleo con *Hyperthreading* hay registros por duplicado, pero por ejemplo la ALU no se replica, por lo que debe ser compartida entre los dos hilos que se estén ejecutando en el caso de que la necesiten.

- **¿Entonces, un núcleo con *Hyperthreading* realiza también multiprogramación?**

Si, cada núcleo puede realizar multiprogramación, y cuando un núcleo está haciendo uso del *Hyperthreading* también está haciendo “multiprogramación” en el sentido de que ciertas unidades de cálculo deben compartirse entre los hilos que estén ejecutando en ese núcleo.

- **¿Los hilos tienen que pertenecer al mismo proceso o pueden ser de procesos diferentes?**

Los hilos pueden pertenecer a procesos monohilo diferentes o a dos hilos de un mismo proceso multihilo.

Por tanto, Hyperthreading es beneficioso cuando la carga de trabajo requiere un procesamiento de tareas que se pueden realizar en paralelo. La edición de video y el renderizado en 3D son ejemplos de tareas que podrían beneficiarse del trabajo de *HyperThreading*. *HyperThreading* también es útil cuando se desea que la CPU ejecute tareas más ligeras, como aplicaciones en segundo plano, mientras que las aplicaciones más intensivas, como juegos, se envían a otro núcleo de procesador en procesadores de múltiples núcleos.

## 10 Modo dual (usuario y *kernel*)

El **código del propio sistema operativo**, es decir, su conjunto de instrucciones para gestionar recursos y proporcionar servicios, se encuentra **codificado en lenguaje máquina al cargarse en memoria principal al arranque del sistema**.

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 interaccionan con el hardware, **por ejemplo en las rutinas ISR**. 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.

Para evitar este tipo de situaciones, los diseñadores han buscado una **solución basada en un modo dual de operación**:

- El **modo núcleo, *kernel*, supervisor o privilegiado** (no tiene nada que ver con el *root*): Este modo está **asociado al procesamiento de instrucciones y rutinas del sistema operativo**. La gestión de recursos del sistema y los servicios que éste ofrece están accesibles solo a través de lo que se llaman **llamadas nativas al sistema**. Las llamadas nativas al sistema son rutinas o funciones a nivel de núcleo, es decir, situadas en la parte de la memoria principal reservada al sistema operativo (no accesible por parte del usuario).  
Por tanto, el núcleo posee las funciones y servicios del sistema que deben protegerse, y que residen en lenguaje máquina en memoria principal.
- El **modo usuario**: Este modo está **asociado al procesamiento de instrucciones y rutinas de los programas de usuario**, es decir, ejecuta rutinas cuyo código máquina y datos está situado en la parte de memoria principal reservada a dichos programas. Un programa de usuario no puede acceder directamente a posiciones de memoria del núcleo del sistema operativo, debe hacerlo a través de la invocación de llamadas al sistema.

De forma general, **¿cuándo se ejecuta o se accede al código del núcleo?**

- 1) Cuando una aplicación ejecuta una **llamada nativa al sistema**, ya sea indirectamente a través de una API (*fork()*, *read()*, *wait()*, etc) – *Wrapper* –, desde invocaciones a ejecutables desde el interprete de comandos, o desde un lenguaje de bajo nivel. Las llamadas nativas al sistema, como se estudiará posteriormente, dan lugar a que se produzca una interrupción especial de acceso al núcleo.
- 2) Cuando un **dispositivo de E/S provoca una interrupción**, ya que hay que **ejecutar las rutinas del núcleo necesarias para tratar esa interrupción**.
- 3) Cuando hay alguna **interrupción por fallo de tipo hardware** que no provoquen el apagado repentino de la máquina.
- 4) Cuando el **planificador debe cambiar de proceso**, por ejemplo al cumplir un **tiempo de reloj** asignado a un proceso o debido a que se necesitan realizar **funciones de gestión de los recursos del sistema** de forma regular.
- 5) Cuando **una aplicación provoca una excepción**, que da lugar a una **interrupción software** a tratar mediante **las rutinas del núcleo necesarias**, que normalmente desembocarán en la terminación del proceso que provocó dicha excepción. Recuerde que una excepción es una



situación de error detectada por la CPU mientras ejecutaba una instrucción, por ejemplo, división por cero, acceso a direcciones de memoria no permitido, violación de permisos, etc.

A continuación se categorizan las **funciones básicas** que normalmente se encuentran dentro del **núcleo del sistema operativo**:

- **Gestión de procesos:**
  - Creación y terminación de procesos (*fork()*, *wait()*, *exit()*, etc).
  - Planificación y activación de procesos.
  - Intercambio de procesos (salvado y restauración).
  - Sincronización de procesos y soporte para comunicación entre procesos (IPC – *InterProcess Communication*). Memoria compartida, tuberías, semáforos, colas, etc.
  - Gestión de los bloques de control de procesos (BCP) y su actualización.
- **Gestión de memoria**
  - Reserva del espacio de direcciones para los procesos a nivel de usuario.
  - *Swapping* o memoria virtual (planificador a medio plazo).
  - Gestión de paginación y segmentación de memoria.
- **Gestión de E/S**
  - Gestión de buffers internos e intermedios.
  - Reserva de canales de E/S y de dispositivos que requieren de su uso por parte de procesos.
  - Manipulación de dispositivos, conectar, desconectar.
  - Comunicaciones, envío y recepción de mensajes, dispositivos remotos.
- **Gestión de los sistemas de ficheros**
  - ext2, ext3, ext4, ntfs, fat, fat32, reiserfs, etc.
- **Funciones de soporte**
  - Gestión de interrupciones, excepciones.
  - Auditoría y monitorización.

Dicho esto, ¿cómo se cambia de modo?:

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**, que como se han comentado anteriormente y de manera general pueden ser: 1) una **llamada nativa al sistema**, 2) un **tipo de interrupción**.

De esta manera el **hardware detecta los errores de violación de los modos** y es el sistema operativo el que se encarga de tratarlos, del tal forma que **si desde un programa de usuario se hace un intento de ejecutar una instrucción privilegiada del núcleo**, el hardware envía una señal de excepción al sistema operativo para que la trate.



La Figura 1.8 muestra gráficamente un cambio de modo ante una llamada nativa al sistema a partir de una llamada desde una API. 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.

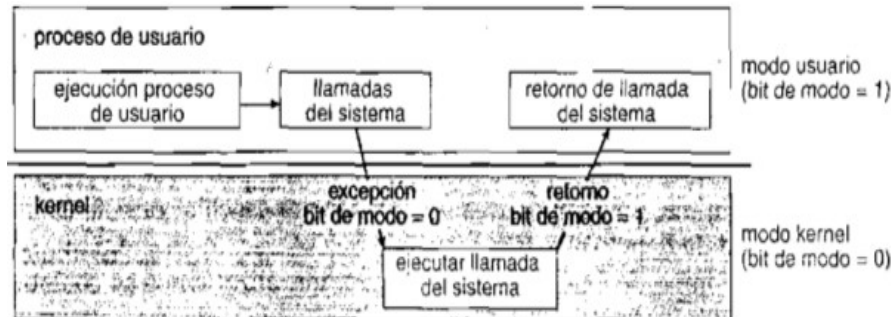


Figura 1.8 Transición de modo usuario a modo kernel.

El SO configura periódicamente una interrupción de reloj vía hardware para evitar perder el control y cambiar a modo núcleo frecuentemente (para realizar tareas de gestión de los recursos del sistema, como por ejemplo tareas de planificación).

## 11 Llamadas nativas al sistema y paso de parámetros

Una llamada nativa al sistema es utilizada por un proceso (programa de usuario que cargado en RAM se convierte en proceso) para **solicitarle un servicio al sistema operativo**, por ejemplo la apertura o cierre de un fichero, o la escritura de un mensaje por la salida estándar del sistema. Normalmente se ejecutan **miles de llamadas nativas al sistema por segundo**, ya sea a partir de procesos de usuario o por el propio sistema operativo para realizar tareas de gestión.

Desde este punto de vista, las llamadas al sistema **proporcionan un interfaz entre un programa o proceso y el núcleo de sistema operativo** para poder invocar los servicios que éste ofrece y gestionar sus recursos, ya que como se ha comentado antes con los modos duales, el usuario no puede acceder o no tiene privilegios directos sobre los recursos que gestiona el sistema operativo.

Los programadores pueden usar **INDIRECTAMENTE** las **llamadas nativas al sistema de varias formas**:

1. Normalmente, los desarrolladores de aplicaciones diseñan e implementan 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. No se tiene por qué saber nada acerca de cómo se implementa dicha función invocada, sino solamente qué es lo que ocurre en su ejecución. Por tanto, la **API oculta al programador la mayor parte de los detalles internos de las llamadas nativas al sistema disponibles**.

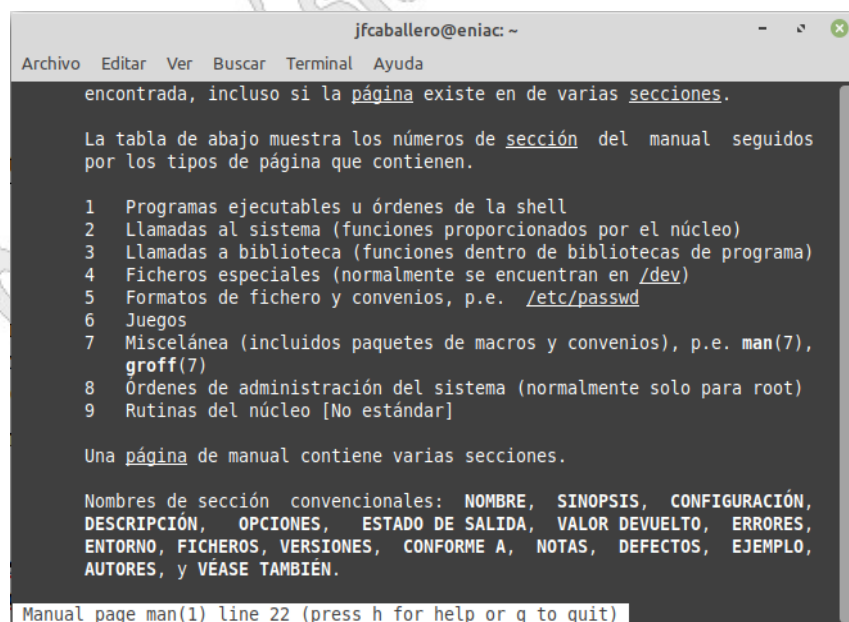
Dos de las API más usuales disponibles para programadores de aplicaciones son la API **Win32** para sistemas Windows y la API de la biblioteca estándar de C, **glibc**, para sistemas basados en POSIX (prácticamente todas las versiones de Gnu/Linux y Mac OS).

En la bibliografía puede encontrar que se nombre “llamada al sistema” a una función de una API, como por ejemplo *printf()* o *write()*, pero se hace para simplificar, ya que realmente eso no son llamadas al sistema. Las funciones que conforman una API de alto nivel invocan a las llamadas nativas al sistema internamente **a través de una función nombrada TRAP**, por lo que realmente actúan como *wrappers* o envoltentes de **funciones nativas del núcleo**.

Por tanto la propia función *printf()* invocada desde un programa en C no es la llamada nativa al sistema en sí, sino que por debajo se encuentra una llamada o un conjunto de llamadas nativas al sistema para realizar la impresión. Como ejemplo, la función para crear nuevos procesos *CreateProcess()* de la API Win32, lo que hace realmente es invocar la llamada nativa al sistema *NTCreateProcess()* del *kernel* de Windows. Lo mismo pasa con la función *write()* de Gnu/Linux, que invoca a una función nativa llamada *\_\_NR\_write* (o indistintamente *sys\_write()*).

2. **Además de las API, en sistemas GNU/Linux hay una función *wrapper* denominada *syscall()*.** Esta función permite invocar llamadas nativas cuando no haya función *wrapper* para ella, aunque es mucho más compleja de usar, ya que requiere indicar una mayor cantidad de parámetro y el identificador de la llamada nativa a invocar. **Puede consultar su uso mediante la orden “*man 2 syscall*” o realizando una búsqueda sobre ello en la web.**

Si ponemos en un terminal la orden “*man man*”, veremos que la **sección 2 de *man*** muestra casi todo el conjunto de llamadas nativas del sistema (más de 300) pero en forma de *wrappers*, para que se puedan utilizar desde C, por tanto no son la llamadas nativas reales, sino el equivalente de una llamada nativa en *glibc*. Por ejemplo, la orden “*man 2 read*” mostraría la llamada *read()* de C que internamente se en cargará de llamar a *\_\_NR\_read* o *sys\_read()* (*\_\_NR\_read* es un entero identificador de llamada nativa y *sys\_read()* el nombre de la llamada nativa).



```
jfcaballero@eniad: ~
Archivo Editar Ver Buscar Terminal Ayuda

encontrada, incluso si la página existe en de varias secciones.

La tabla de abajo muestra los números de sección del manual seguidos
por los tipos de página que contienen.

1 Programas ejecutables u órdenes de la shell
2 Llamadas al sistema (funciones proporcionados por el núcleo)
3 Llamadas a biblioteca (funciones dentro de bibliotecas de programa)
4 Archivos especiales (normalmente se encuentran en /dev)
5 Formatos de fichero y convenios, p.e. /etc/passwd
6 Juegos
7 Miscelánea (incluidos paquetes de macros y convenios), p.e. man(7),
  groff(7)
8 Órdenes de administración del sistema (normalmente solo para root)
9 Rutinas del núcleo [No estándar]

Una página de manual contiene varias secciones.

Nombres de sección convencionales: NOMBRE, SINOPSIS, CONFIGURACIÓN,
DESCRIPCIÓN, OPCIONES, ESTADO DE SALIDA, VALOR DEVUELTO, ERRORES,
ENTORNO, FICHEROS, VERSIONES, CONFORME A, NOTAS, DEFECTOS, EJEMPLO,
AUTORES, y VÉASE TAMBIÉN.

Manual page man(1) line 22 (press h for help or q to quit)
```

Esto es una manera más sencilla de que el usuario pueda invocar a más alto nivel llamadas nativas al sistema y no lo tenga que hacer mediante la función `syscall()` de C, e indicando como parámetro, entre otros, el número de llamada nativa. Las llamadas nativas y su número asociado se encuentran en el fichero `"/usr/include/asm-generic/unistd.h"` (la ruta depende de la versión del núcleo de Linux y de la arquitectura del procesador).

Por otro lado, la **sección 3 de *man*** documenta las funciones que tiene el estándar de C y el estándar Posix y que implementa *glibc*. En *glibc*, además de los *wrappers* que tienen un nombre similar a algunas llamadas nativas que se muestran en la sección 2 de *man*, habrá otras funciones que harán las operaciones para las que estén destinadas, y que para ello invocaran o no a una o varias llamadas nativas al sistema de manera interna. Todas estas funciones (en torno a 2000 funciones), según la literatura que se lea, se nombran también como "llamadas al sistema", pero no lo son, son *wrappers*.

Con el comando `"man 2 intro"` y `"man 3 intro"` puede consultar la descripción de lo que hay documentado en esas secciones de *man*.

3. En **lenguajes de más bajo nivel, como ensamblador**, la invocación a una llamada nativa al sistema requiere de una programación más compleja, **copiando previamente a registros del procesador (mediante instrucciones – nemónicos) cosas como el identificador de la función nativa a utilizar y sus parámetros**, y pasando posteriormente a ejecución en modo núcleo, lo que permitirá ya invocar a la llamada nativa. Estas son 3 instrucciones que hacen cambiar de **modo usuario a modo núcleo** según diferentes hardware: *INT 0X80*, *SYSENTER*, *SYSCALL*. Esta es la manera más "directa" de las tres descritas para invocar a una llamada nativa del sistema, pero la más compleja.

Este tipo de programación la suelen hacer los **diseñadores y programadores del núcleo del sistema operativo**, siendo poco frecuente en el programador habitual a nivel de usuario. En GNU/Linux, los ficheros programados en **lenguaje ensamblador** tienen la **extensión .S** o **extensión .asm**

A continuación se muestra un ejemplo de invocación en ensamblador de la llamada nativa `sys_exit()` y `sys_write()`. Su equivalente a nivel de API (wrapper) son `exit()` y `write()`:

## Linux System Calls

You can make use of Linux system calls in your assembly programs. You need to take the following steps for using Linux system calls in your program –

- Put the system call number in the EAX register.
- Store the arguments to the system call in the registers EBX, ECX, etc.
- Call the relevant interrupt (80h).
- The result is usually returned in the EAX register.

There are six registers that store the arguments of the system call used. These are the EBX, ECX, EDX, ESI, EDI, and EBP. These registers take the consecutive arguments, starting with the EBX register. If there are more than six arguments, then the memory location of the first argument is stored in the EBX register.

The following code snippet shows the use of the system call `sys_exit` –

```
mov    eax,1      ; system call number (sys_exit)
int    0x80       ; call kernel
```

The following code snippet shows the use of the system call `sys_write` –

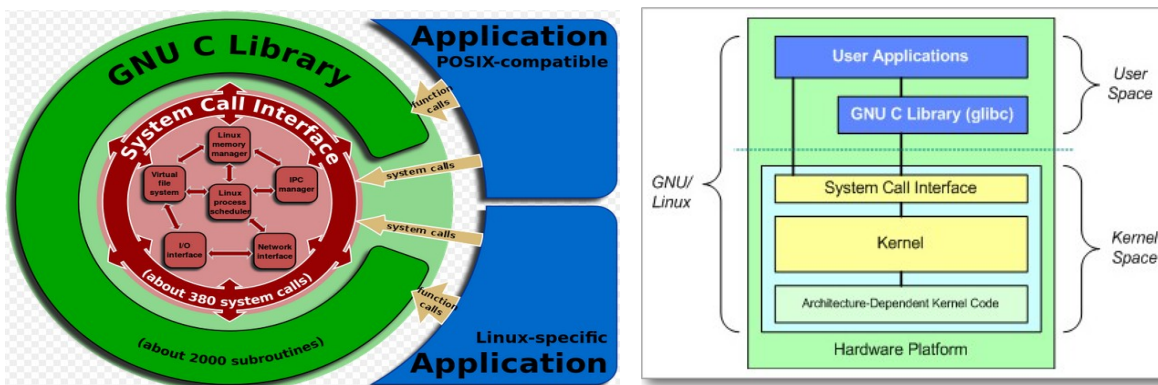
```
mov    edx,4      ; message length
mov    ecx,msg     ; message to write
mov    ebx,1      ; file descriptor (stdout)
mov    eax,4      ; system call number (sys_write)
int    0x80       ; call kernel
```

All the syscalls are listed in `/usr/include/asm/unistd.h`, together with their numbers (the value to put in EAX before you call `int 80h`).

The following table shows some of the system calls used in this tutorial –

%eax	Name	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	int	-	-	-	-
2	sys_fork	struct pt_regs	-	-	-	-
3	sys_read	unsigned int	char *	size_t	-	-
4	sys_write	unsigned int	const char *	size_t	-	-

Las siguientes figuras muestran gráficamente dónde se sitúan las llamadas nativas al sistema y las “llamadas al sistema” a través de API:



Habitualmente, cada **llamada nativa al sistema** tiene asociado un **número identificador** (se mantiene constante entre diferentes versiones del *kernel* para ofrecer compatibilidad) que hace referencia a la **tabla de llamadas al sistema**. La tabla de llamadas al sistema es un **array de punteros a función**, se sitúa en la parte de la memoria reservada al **núcleo del sistema operativo** y contiene **punteros a llamadas nativas del sistema operativo**. El **índice** que cada elemento de la tabla indica la **identificación de la llamada nativa** y contiene la dirección de comienzo de la misma en memoria.

Cuando se invoca a una llamada nativa al sistema hay que hacerlo mediante la invocación de funciones especiales denominadas de tipo **TRAP**. Este tipo de funciones se invocarían desde el interior de una función API (*wrapper*), por ejemplo desde dentro de la función *getpid()*. Posteriormente se verá este proceso con algo más de detalle.

Cuando se realiza una **llamada nativa al sistema** puede ser necesario **pasar parámetros al núcleo** del sistema operativo. De manera general se emplean dos métodos:

1. El más sencillo de ellos consiste en pasar los parámetros a una serie de **registros del procesador que son accesibles en modo usuario** y que posteriormente serán copiados al núcleo. Recuerde que a nivel de usuario no se puede acceder directamente a los registros del procesador no visibles al mismo, ni tampoco a la memoria del kernel.

Tiene la desventaja de que pudiera haber más parámetros que registros disponibles, por lo que esta opción no sería viable. Para estos casos cabe la opción de que **los parámetros se almacenen en un bloque en memoria a nivel de usuario**, y la **dirección del bloque** se pasa como parámetro a un registro del procesador visible al usuario, que posteriormente será copiado al núcleo.

2. Otra alternativa sería que el proceso de usuario colocase los parámetros en la **pila de memoria de dicho proceso**, y el sistema operativo se encargará de extraer de la pila esos parámetros y **copiarlos al núcleo**.

### **11.1 Pasos generales en la invocación de una llamada nativa al sistema mediante funciones de tipo TRAP**

Una vez presentado el concepto de llamada nativa al sistema, a continuación se muestra qué ocurre cuando se produce una invocación y cómo realizarla.

Se mostrará **el proceso general** mediante el uso de APIs, que puede variar entre versiones de los diferentes sistemas:

1. Cuando se produce una llamada al sistema a partir de una API (*wrapper*), los **parámetros asociados** a la misma se cargarían en la **pila del proceso a nivel de usuario**, o en **registros accesibles** a nivel de usuario, dependiendo del sistema. Además de ello, se copia en un registro del procesador el identificador de la llamada nativa que se desea invocar (registro *eax* en un ejemplo más adelante). En este paso se está todavía en modo usuario.
2. Posteriormente, la propia función API invocada ejecuta también otra función especial denominada de tipo **trap** (cambiará de nombre según el sistema). Estas funciones tipo *trap* no forman parte de la API como tal, sino que son **las funciones de la propia API las que la invocan internamente** (todavía en modo usuario).

La función de tipo **trap**, entre otras cosas, se encargará de ejecutar una **interrupción**



**especial** que conllevará a un **cambio de modo** (*INT 0X80* en un ejemplo más adelante), de modo usuario a modo núcleo. Este cambio de modo se realiza por hardware una vez se invoca a esa interrupción especial.

Ya en modo núcleo se **copia el identificador de llamada nativa a invocar** (desde el registro *eax* en nuestro ejemplo), y los **parámetros almacenados por la llamada a nivel de API**.

3. Siempre que se realiza una llamada al sistema hay que **guardar el estado** (salvado de contexto) de ejecución actual y el valor del PC, ya que se va a saltar a una zona de memoria donde se encontrará alojada la llamada nativa del sistema a invocar y después no se sabría volver al proceso que se estaba ejecutando. Para ello se siguen invocando otras rutinas para hacer estas operaciones de salvado, justo **antes de empezar a ejecutar la llamada nativa al sistema** concreta a partir de su número de identificación.
4. Posteriormente al salvado se **examina el identificador y los parámetros correspondientes a la llamada nativa a invocar** y se busca en una tabla o vector de rutinas si existe dicho número de identificación y si los parámetros son correctos.
5. Tras identificar que existe una rutina para la llamada nativa realizada y que los parámetros son correctos, **se procede a ejecutarla**. La **tabla de llamadas nativas al sistema** contiene la **dirección del lugar del núcleo donde se encuentra** la llamada nativa a ejecutar.
6. Una vez ejecutada la llamada nativa al sistema se invocan otras subrutinas denominadas de tipo **RETURN FROM TRAP**, para:
  - **Devolver el parámetro de retorno** que de la función nativa (se hace machacando el contenido del registro *eax* que contenía el identificador de la llamada nativa).
  - **Restaurar el contexto** del proceso salvado, el que realizó la llamada al sistema a través de API (o de otro proceso según la política de planificación que exista).
  - **Copiar en la pila** del proceso que invocó la llamada nativa (*wrapper* desde API) el **valor del registro *eax***.
  - **Pasar de modo núcleo a modo usuario**.
7. Cuando finalizan esas acciones se regresa el control a la función de API (*printf()*, *write()*, etc) y **a nivel de usuario se descarga la pila** y se comprueba el resultado de la ejecución de la petición al sistema.

Tenga en cuenta que el **esquema proporcionado es genérico** y que no se ha tenido en cuenta que la ejecución de la llamada nativa al sistema se pueda interrumpir. Dependiendo del sistema y su configuración o políticas del núcleo, **una llamada nativa al sistema se podrían ver interrumpida por algún evento o señal**. Si el sistema admite ejecutar interrupciones en una llamada nativa, debe tener implementado en su núcleo determinadas instrucciones y rutinas que lo contemplen.

## 11.2 Ejemplo de llamadas nativas al sistema en sistema GNU/Linux clásico

A continuación se muestra cómo se lleva a cabo el mecanismo mediante el cual GNU/Linux implementa las **llamadas nativas al sistema en una arquitectura clásica x86 (32 bits)**. Tenga en cuenta que dependiendo de la arquitectura del procesador y dependiendo de la versión del sistema operativo y su kernel, la interfaz de llamadas nativas al sistema puede variar, así como el procedimiento llevado a cabo para realizarlo.

Para la arquitectura y versión de núcleo comentada, en el fichero *include/asm-i386/unistd.h* que aparece en la siguiente figura (cuidado, hay varios ficheros nombrados así GNU/Linux) están **identificadas las llamadas nativas al sistema**, que se ofrecen con su correspondiente número. Por ejemplo `__NR_close` corresponde a la llamada nativa al sistema `sys_close()` (lo usaremos indistintamente), que es la función `close()` a nivel de API. En un sistema más actual ese fichero se podría situar en la ruta *usr/include/asm-generic/unistd.h*.

Cada sistema operativo usa el hardware de la computadora de una manera específica. Por tanto, para un mismo hardware, dos sistemas operativos distintos mostrarán un interfaz de llamadas nativas al sistema distinto.

```

/*
 * This file contains the system call numbers.
 */

#define __NR_restart_syscall    (__NR_SYSCALL_BASE+ 0)
#define __NR_exit               (__NR_SYSCALL_BASE+ 1)
#define __NR_fork               (__NR_SYSCALL_BASE+ 2)
#define __NR_read               (__NR_SYSCALL_BASE+ 3)
#define __NR_write              (__NR_SYSCALL_BASE+ 4)
#define __NR_open               (__NR_SYSCALL_BASE+ 5)
#define __NR_close              (__NR_SYSCALL_BASE+ 6)
/* 7 was sys_waitpid */
#define __NR_creat              (__NR_SYSCALL_BASE+ 8)
#define __NR_link               (__NR_SYSCALL_BASE+ 9)
#define __NR_unlink             (__NR_SYSCALL_BASE+ 10)
#define __NR_execve             (__NR_SYSCALL_BASE+ 11)
#define __NR_chdir              (__NR_SYSCALL_BASE+ 12)
#define __NR_time               (__NR_SYSCALL_BASE+ 13)
#define __NR_mknod              (__NR_SYSCALL_BASE+ 14)
#define __NR_chmod              (__NR_SYSCALL_BASE+ 15)
#define __NR_lchown             (__NR_SYSCALL_BASE+ 16)
/* 17 was sys_break */
/* 18 was sys_stat */

... mas llamadas al sistema ...

#define __NR_pipe2              (__NR_SYSCALL_BASE+359)
#define __NR_inotify_init1      (__NR_SYSCALL_BASE+360)
#define __NR_preadv             (__NR_SYSCALL_BASE+361)
#define __NR_pwritev            (__NR_SYSCALL_BASE+362)
#define __NR_rt_tgsigqueueinfo  (__NR_SYSCALL_BASE+363)
#define __NR_perf_event_open    (__NR_SYSCALL_BASE+364)

```

*Fichero unistd.h de un kernel típico 2.6*

A nivel de **lenguaje ensamblador**, para una **versión 2.6 del kernel de Linux** y para una **arquitectura x86**, se utiliza la invocación de una interrupción especial **INT 0x80 como manera de comenzar el proceso de acceso a una llamada nativa al sistema y cambiar a modo núcleo**. Los sistemas más modernos utilizan otra manera de acceder al núcleo mediante la invocación de la

instrucción **SYSCALL** (o incluso la instrucción **SYSENTER** en otros núcleos).

**int 0x80**: Interrupción software especial que iniciará el proceso de acceso a la llamada nativa al sistema que se vaya a invocar:

- Esta interrupción **se invoca a partir de la rutina `__init trap_init()`**, que se encuentra en el fichero **`arch/x86/kernel/traps.c`**
- El número de la interrupción software **`int 0x80`** está definido por la constante **`SYSCALL_VECTOR`**, que se encuentra definida en el fichero **`arch/x86/include/asm/irq_vectors.h`**

Dicho esto y resumiendo, en el código de la función **`__init trap_init()`** se establece el método de entrada al núcleo mediante la función

**`set_system_gate (SYSCALL_VECTOR,&system_call),`**

que invocará a la interrupción **`int 0x80`** y se producirá un salto a la zona de memoria del kernel donde se encuentra la función **`system_call()`**, situada en el fichero en ensamblador **`arch/x86/kernel/entry_32.S`**.

**`__init trap_init()` → `set_system_gate (SYSCALL_VECTOR,&system_call)` → `int 0x80` → `system_call()`**

Recuerde que de manera general y antes de entrar en modo núcleo se habrán hecho otras operaciones como por ejemplo almacenar en el registro **`%eax`** del procesador el identificador de la llamada nativa a invocar.

```

arch/i386/kernel/traps.c [995-996,1009-1017,1032,1037-1040]
995 void __init trap_init(void)
996 {
...
1009     set_trap_gate(0,&divide_error);
1010     set_intr_gate(1,&debug);
1011     set_intr_gate(2,&nmi);
1012     set_system_intr_gate(3, &int3); /* int3-5 can be called from all */
1013     set_system_gate(4,&overflow);
1014     set_system_gate(5,&bounds);
1015     set_trap_gate(6,&invalid_op);
1016     set_trap_gate(7,&device_not_available);
1017     set_task_gate(8,GDT_ENTRY_DOUBLEFAULT_TSS);
...
1032     set_system_gate(SYSCALL_VECTOR,&system_call);
...
1037     cpu_init();
1038
1039     trap_init_hook();
1040 }

```

Para intentar dar una mejor comprensión del complejo proceso de llamadas nativas al sistema y aligerar la lectura, se omiten algunas instrucciones, ejecución de subrutinas y otras comprobaciones necesarias para ello.

En la llamada **`set_system_gate (SYSCALL_VECTOR,&system_call)`** se produce un cambio a modo núcleo con **`int 0x80`** y se dan los siguientes pasos:

1. La función `system_call()` copia el código **identificador de la llamada nativa al sistema** que se quiere invocar, recogido del registro `%eax` del procesador, que se habrá cargado por la función `wrapper` en algún paso anterior a la invocación de `set_system_gate(SYSCALL_VECTOR, &system_call)`, cuando todavía se estaba en modo usuario (ahora se está en modo núcleo). También se copia otra información como los **parámetros de entrada** que tendrá que usar la función nativa.
2. Acto seguido la función `system_call()` también guarda varios **registros del procesador** con la macro `SAVE_ALL` para la futura restauración del contexto del proceso de usuario interrumpido por `INT 0x80`. Esta macro está situada en el archivo en ensamblador `arch/x86/kernel/entry_32.S`.
3. Se comprueba en la tabla de llamadas al sistema que el número de llamada nativa pedido es válido mediante el código situado bajo las etiquetas `"syscall_trace_entry:"`.

1. Si no es válido, se ejecuta el código bajo la etiqueta `"syscall_badsys:"`, situada también en el archivo en ensamblador `arch/x86/kernel/entry_32.S`. Este código devuelve el código de error `ENOSYS` para ponerlo en `errno` y el registro `%eax` se pone a -1 (dicho registro tenía hasta ese momento el número identificativo de la llamada nativa al sistema a invocar), para terminar saltando al código de la subrutina `resume_userspace()`, que se comenta a continuación.
2. Si es válido se ejecuta el código bajo la etiqueta `"syscall_call:"`. Aquí se invoca la llamada nativa mediante la rutina `sys_call_table()` (los parámetros de la llamada nativa ya se copiaron anteriormente al espacio del núcleo) y se guarda el **valor de retorno** en el registro `%eax`. Dicho registro tenía hasta ese momento el número identificativo de la llamada nativa al sistema a invocar.

Al finalizar la ejecución de la llamada nativa al sistema se ejecuta el código que se encuentra bajo la etiqueta `"syscall_exit:"`. Aquí se realizan una serie de comprobaciones para ver si hay alguna interrupción pendiente en el sistema con algún trabajo más prioritario. **Si no hay nada más prioritario** se ejecuta el código situado bajo la etiqueta `"syscall_exit_work:"`, que terminará saltando al código de la subrutina `resume_userspace()`, que se comenta a continuación. No entraremos en este ejemplo en el proceso que se realiza si hubiera algo más prioritario.

`"syscall_call:"` → `sys_call_table()` → `"syscall_exit:"` → `"syscall_exit_work:"`  
→ `resume_userspace()`

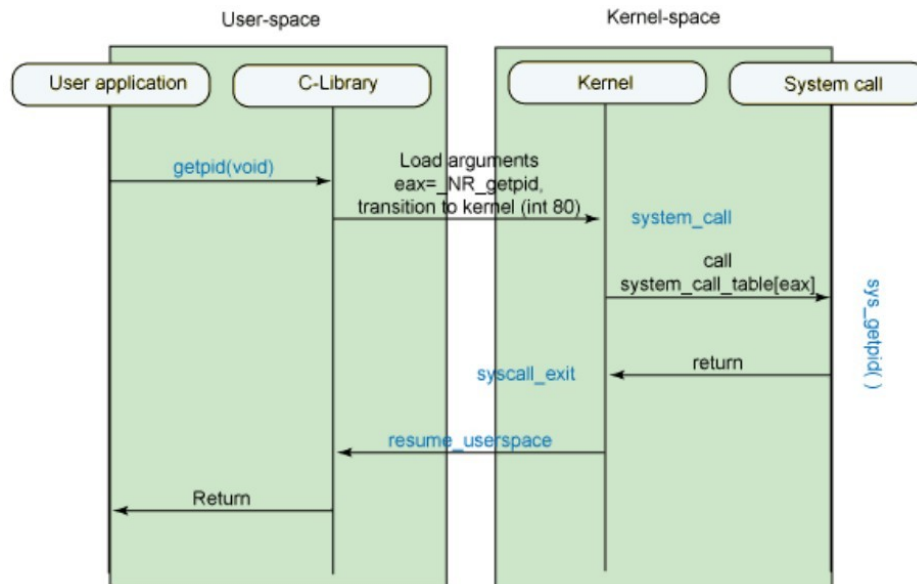
4. Para volver a modo usuario y **poder restaurar y seguir a partir de la llamada al sistema a nivel de API** se ejecuta el código de la subrutina `resume_userspace()`, que a su vez ejecuta el código bajo la etiqueta `"restore_all:"`.

Bajo `"restore_all:"` se restauran los registros almacenados previamente con `SAVE_ALL` y los valores necesarios en la pila a nivel de usuario, incluido el valor devuelto por la función, que está almacenado en `%eax`, y se cambia a modo usuario mediante instrucciones denominadas de tipo `IRET`.

5. Si la llamada de API fue exitosa la biblioteca a nivel de usuario (`glibc`) obtiene el resultado de la pila y seguirá la ejecución del proceso de usuario por donde iba.

6. Si la llamada no tuvo éxito o existe algún tipo de error, el valor devuelto a la pila a través del registro `%eax` es negativo (`-1`), y como programadores se debería consultar a nivel de usuario la variable global `errno` (contiene el código de error).

En la siguiente figura se muestra un resumen gráfico del proceso exitoso de una llamada nativa al sistema desde una API:



### 11.3 Localizar cuáles son los números (IDs) de llamadas nativas al sistema en un SO GNU/Linux actual

El mecanismo mediante el cual GNU/Linux implementa las llamadas nativas al sistema varía según la **arquitectura de la máquina** y su **versión del kernel**. Se ha mostrado el procedimiento para una arquitectura clásica Intel x86 (32 bits) con kernel 2.6 de Linux.

Adicionalmente, podemos ver este tipo de información sobre nuestro sistema con los siguientes comandos en la terminal:

- `uname -a` para información sobre la versión del kernel.
- `lsb_release -cd` para información sobre la versión del sistema operativo.
- `lscpu` para información extensa de la CPU.

```
jfcaballero@eniac: ~
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
jfcaballero@eniac:~$ uname -a
Linux eniac 5.11.0-37-generic #41~20.04.2-Ubuntu SMP Fri Sep 24 09:06:38
UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
jfcaballero@eniac:~$ lsb_release -cd
Distributor ID: Linuxmint
Description:    Linux Mint 20.2
Release:        20.2
Codename:       uma
jfcaballero@eniac:~$
```



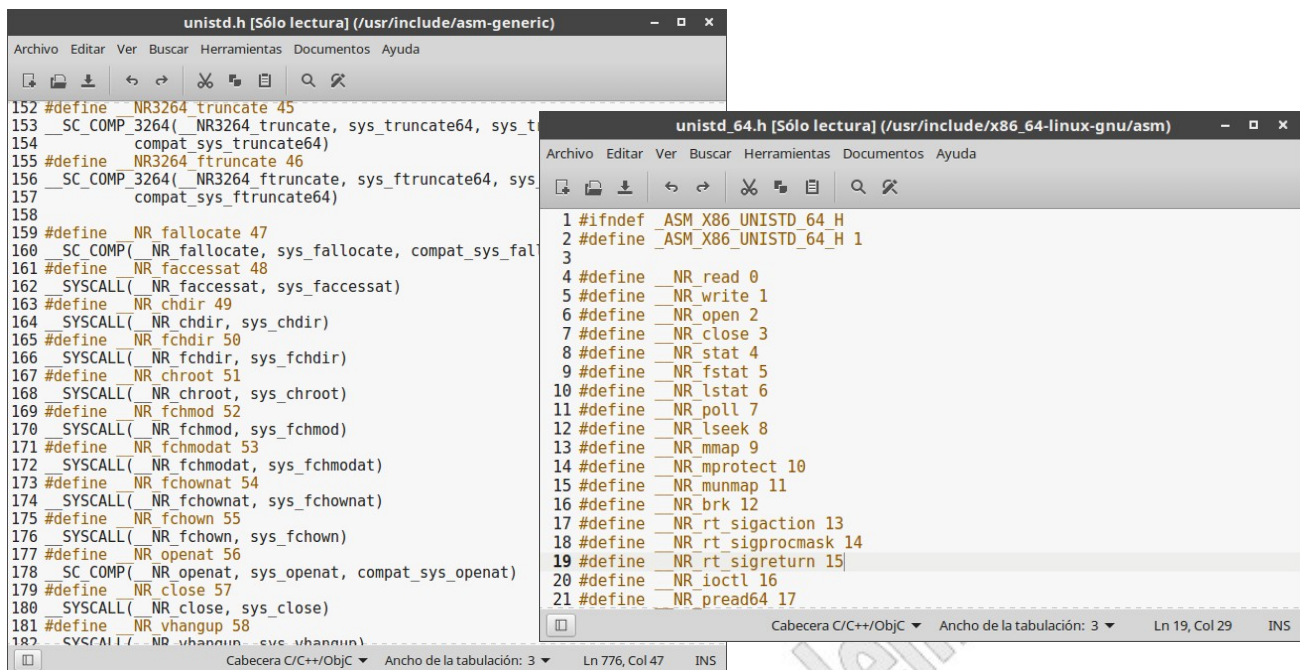
```
jfcaballero@eniac: ~
Archivo Editar Ver Buscar Terminal Ayuda
jfcaballero@eniac:~$ lscpu
Arquitectura: x86_64
modo(s) de operación de las CPUs: 32-bit, 64-bit
Orden de los bytes: Little Endian
Address sizes: 39 bits physical, 48 bits virtual
CPU(s): 8
Lista de la(s) CPU(s) en línea: 0-7
Hilo(s) de procesamiento por núcleo: 2
Núcleo(s) por «socket»: 4
«Socket(s)»: 1
Modo(s) NUMA: 1
ID de fabricante: GenuineIntel
Familia de CPU: 6
Modelo: 140
Nombre del modelo: 11th Gen Intel(R) Core(TM) i7-1165G7
@ 2.80GHz
Revisión: 1
CPU MHz: 1809.276
```

La cuestión ahora es encontrar el fichero *unistd.h*, que es donde están listadas las identificaciones de las llamadas nativas al sistema. Para ello, viene bien usar el comando **locate** en una terminal de Linux. Si no lo tenemos instalado, basta con hacer **sudo apt-get install locate** para instalarlo, seguido de **sudo updatedb**.

Posteriormente bastará con poner **locate unistd.h** y veremos una larga lista de los *paths* de todos los ficheros con ese nombre exacto en nuestro sistema. Nos fijaremos en algunos los subrayados de la siguiente imagen:

```
jfcaballero@eniac ~
Archivo Editar Ver Buscar Terminal Ayuda
/snap/mathpix-snipping-tool/40/usr/include/unistd.h
/snap/mathpix-snipping-tool/40/usr/include/asm-generic/unistd.h
/snap/mathpix-snipping-tool/40/usr/include/linux/unistd.h
/snap/mathpix-snipping-tool/40/usr/include/x86_64-linux-gnu/asm/unistd.h
/snap/mathpix-snipping-tool/40/usr/include/x86_64-linux-gnu/bits/unistd.h
/snap/mathpix-snipping-tool/40/usr/include/x86_64-linux-gnu/sys/unistd.h
/usr/include/unistd.h
/usr/include/asm-generic/unistd.h
/usr/include/linux/unistd.h
/usr/include/x86_64-linux-gnu/asm/unistd.h
/usr/include/x86_64-linux-gnu/bits/unistd.h
/usr/include/x86_64-linux-gnu/sys/unistd.h
/usr/local/MATLAB/R2017a/polyspace/verifier/cxx/cinclude/_polyspace_unistd.h
/usr/local/MATLAB/R2017a/polyspace/verifier/cxx/include/include-libc/unistd.h
/usr/local/MATLAB/R2017a/polyspace/verifier/cxx/include/include-libc/asm/unistd.h
```

- **/usr/include/asm-generic/unistd.h** es donde se halla la lista de las llamadas nativas al sistema con su correspondiente número, y que como programadores deberíamos utilizar.
- **/usr/include/linux/unistd.h** apunta a **/usr/include/x86\_64-linux-gnu/asm/unistd.h**, el cual devuelve un puntero a **/usr/include/x86\_64-linux-gnu/asm/unistd\_32.h** o a **/usr/include/x86\_64-linux-gnu/asm/unistd\_64.h**, es decir, dependiendo de la arquitectura. El usar una u otra el cosa del ensamblador al detectar la arquitectura, el programador usará las identificaciones de **/usr/include/asm-generic/unistd.h**.



- **/usr/include/unistd.h** contiene macros y constantes simbólicas que usamos con muchas funciones a nivel de API que implementa POSIX y que tienen que ver con la gestión de procesos.
- **/usr/include/x86\_64-linux-gnu/bits/unistd.h** es una colección de macros que usa **/usr/include/unistd.h**.

## 12 Interpretador 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*.

La función principal del intérprete de comandos es *obtener y ejecutar un 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.

Los sistemas basados en GNU/Linux y otros sistemas operativos implementan 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 GNU/Linux `"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 añadan, lo

que hace es buscar ese comando en el directorio desde donde se hace una invocación y en la variable ***PATH*** del sistema.

## 13 Un poco de historia

En Moodle dispone de material interesante ligado a la historia de los Sistemas Operativos.

### Bibliografía

- 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. Gagne, 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.