El sistema operativo: una máquina virtual

José Ramón Herrero Zaragoza
Josep Lluís Marzo i Lázaro
Enric Morancho Llena

PID_00214804

UOC
Universitat Oberta de Catalunya

Índice

Introducción

En este módulo didáctico, afianzamos el concepto de sistema operativo, introducido en el módulo anterior, e introducimos el concepto de máquina virtual. Asimismo, explicamos los diferentes mecanismos que provocan la ejecución de código propio del sistema operativo: los directos (llamadas al sistema, interrupciones y excepciones), y los indirectos (bibliotecas y aplicaciones de sistema).

También enumeramos los requisitos que tiene que cumplir el hardware para poder instalar un sistema operativo multiusuario capaz de proteger a cada usuario del resto de usuarios.

Finalmente, hacemos una breve introducción a la planificación del procesador.

Objetivos

Los materiales didácticos de este módulo contienen las herramientas necesarias para alcanzar los objetivos siguientes:

1. Afianzar el concepto de sistema operativo introducido en el módulo anterior.

2. Adquirir el concepto de máquina virtual.

3. Comprender las diferencias entre los diferentes mecanismos directos de entrada al sistema operativo.

4. Conocer los mecanismos indirectos de entrada al sistema operativo.

5. Ser consciente de los requisitos que tiene que cumplir el hardware para poder instalar un sistema operativo multiusuario capaz de proteger a cada usuario del resto de usuarios.

6. Entender el funcionamiento básico de la planificación del procesador.

1. Software de sistema y máquinas virtuales

1.1. El concepto de máquina virtual

La finalidad básica del sistema operativo es esconder el **hardware** del sistema computador mediante una capa de software, por medio de la cual accedemos al hardware para trabajar. El sistema operativo define una **máquina virtual** en la que los usuarios pueden trabajar de una manera más cómoda de la que lo harían si trabajaran directamente con los elementos que componen el sistema computador.

Esta máquina virtual es muy conveniente para los usuarios programadores porque les ofrece una visión idealizada del sistema computador. Los programadores no tendrán que preocuparse de los detalles específicos del hardware (como por ejemplo el tipo de teclado o el formato del disco duro) y podrán centrarse en la programación de los aspectos directamente relacionados con el problema que tienen que resolver. Esta máquina virtual está definida por la interfaz de llamadas ofrecidas por el sistema operativo. Este último también tiene la importante tarea de transformar las abstracciones ofrecidas (como ficheros) en referencias físicas concretas (como pistas en el disco duro).

Hay muchas maneras de esconder este hardware y agrupar y organizar las funciones que tendrá la nueva máquina. Cada una de estas combinaciones determina un tipo diferente de sistema operativo. Desde el punto de vista del usuario que accede a los recursos ofrecidos por la máquina, estos recursos vienen dados por el entorno de acceso; este entorno está determinado por el sistema operativo y no por el hardware.

El hecho de ocultar el hardware a los usuarios y a los programadores tiene, entre otros, dos objetivos principales, que son los siguientes:

- La abstracción del sistema computador como un hardware determinado para tener una visión más global y sencilla de la máquina.

- Proporcionar un modo de funcionamiento nuevo y más seguro cuando se ejecuta el código que pertenece al sistema operativo. Este aspecto es muy importante para evitar que los usuarios tengan acceso a ciertos elementos del sistema que podrían afectar al funcionamiento normal y que de alguna manera tienen que estar supervisados.

Ahora bien, los usuarios finales del sistema computador, que básicamente desean ejecutar programas, pueden encontrar que la máquina virtual ofrecida por el sistema operativo se encuentra en un nivel demasiado bajo. Para facili-

tar la tarea de estos usuarios, el resto de software de sistema también les ofrece una máquina virtual a un nivel mucho más alto, que es el **intérprete de comandos** (también llamado intérprete de órdenes o *shell*).

El intérprete de comandos es el programa más habitual que configura la interfaz con la que dialoga el usuario. Para simplificarlo, podemos decir que el intérprete de comandos es el interlocutor entre los usuarios y el sistema operativo.

En función del tipo de usuario, este intérprete puede estar más o menos especializado o puede ser más o menos restrictivo. Por ejemplo, un programador que tiene que desempeñar tareas de desarrollo o de gestión del sistema tendrá un intérprete de comandos genérico que le permitirá hacer muchas acciones diferentes; en cambio, un administrativo puede tener un intérprete de comandos muy restrictivo que sólo le permita ejecutar las aplicaciones propias de su tarea.

1.2. Capas dentro del software de sistema

En la figura 1, podemos ver las capas del software de sistema con más detalle. Dentro del software de sistema distinguimos tres capas, que son el **sistema operativo** (núcleo), las **bibliotecas del sistema** y las **aplicaciones de sistema** (entre ellas, el intérprete de pedidos). Todos los niveles contienen código útil y accesible para los usuarios, aunque el acceso es diferente en cada nivel.

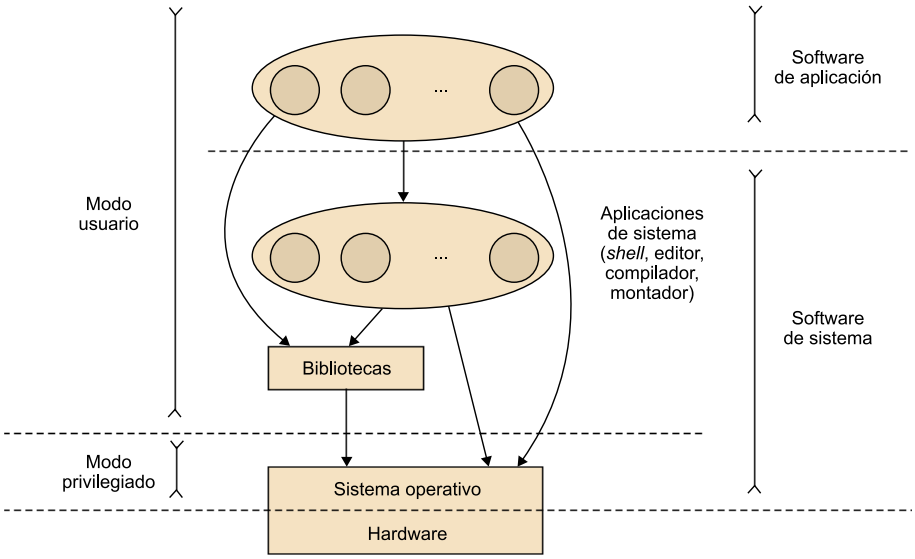

Figura 1. Clasificación de los tipos de software existentes en un sistema computador

La manera de acceder a estos niveles del software de sistema es la siguiente:

1) El **sistema operativo** es la capa que se encuentra más cerca del hardware y contiene las rutinas de gestión del sistema más relacionadas con sus recursos físicos. Estas rutinas se pueden utilizar por medio de las llamadas al sistema.

Para garantizar que únicamente el código del sistema operativo pueda acceder a determinados recursos del sistema computador, este código se ejecutará en una modalidad de ejecución privilegiada; el código correspondiente a las otras capas se ejecutará en una modalidad de ejecución no privilegiada (modo de ejecución usuario).

2) En un nivel superior del sistema operativo, encontramos las **bibliotecas de rutinas**. Estas bibliotecas ofrecen a los programadores dos tipos de rutinas; algunas son de uso muy común, pero no forman parte del sistema operativo (bibliotecas del lenguaje), mientras que otras permiten a los programadores invocar llamadas al sistema de forma independiente del lenguaje máquina (bibliotecas del sistema).

3) El **intérprete de comandos** es accesible mediante la interfaz de usuario (en modo texto o en modo gráfico). En este nivel, el sistema espera la introducción de pedidos de manera interactiva por parte de usuarios reales o bien por parte de programas formados por grupos de pedidos (estos programas reciben el nombre de *scripts* o *shellscripts*), que son interpretados por el intérprete de comandos.

1.2.1. Las llamadas al sistema

Como hemos dicho, un sistema operativo proporciona un entorno para la ejecución de programas. Una parte de este entorno consiste en un conjunto de servicios que son accesibles por los procesos[1] que se están ejecutando en el sistema. Estas funcionalidades son como una extensión de las instrucciones que puede ejecutar el procesador y están en la línea de una concepción del sistema como una máquina virtual que puede hacer cosas más complicadas que las que puede hacer el hardware en sí mismo.

(1)Tal como introdujimos en el módulo anterior, un proceso no es más que un programa en ejecución.

Los programadores de aplicaciones invocan los servicios del sistema operativo desde sus programas haciendo llamadas al sistema[2].

(2)Estas llamadas al sistema también se llaman API (Application Programming Interface) del sistema operativo.

Aunque cada sistema operativo puede ofrecer servicios muy específicos, de manera general podemos agruparlos en las clases siguientes:

1) Gestión de procesos. En este apartado, englobamos las llamadas que hacen referencia a la creación y la eliminación de procesos, pero también podemos encontrar otros servicios, como la suspensión y la reanimación de procesos. También podemos modificar atributos de ejecución, como la prioridad o el tiempo de cuota de CPU asignado. Tal como veremos más adelante, todas estas llamadas se encuentran bajo el control del núcleo del sistema operativo, que determina en última instancia si la petición realizada por un proceso puede ser servida de acuerdo con sus privilegios.

2) Señalización entre procesos. En un sistema multitarea, el nexo natural de unión entre todas las aplicaciones en ejecución es el sistema operativo. Así, el vehículo apropiado para comunicar y sincronizar procesos es el núcleo del sistema operativo. Señales, semáforos, regiones críticas y otras funciones son ejemplos concretos de esta clase de servicios.

3) Gestión de dispositivos de entrada/salida. Los servicios relacionados con el sistema de entrada/salida son de los más completos y también de los más utilizados. Tienen como funciones principales crear canales de entrada/salida[3], abrirlos y cerrarlos, así como escribirlos y leerlos. Los sistemas operativos también proporcionan un conjunto de llamadas para conocer el estado de un canal o cambiarle los atributos.

(3)Los canales básicamente permiten extraer datos desde un proceso o hacia éste.

4) Gestión directa de los recursos del sistema. Este grupo de servicios es muy específico de cada sistema operativo, pero un ejemplo común es la gestión de la memoria. Algunas de las operaciones que se efectúan con la memoria del sistema son capturar, liberar, proteger y compartir.

5) Gestión del sistema de archivos. Junto con el bloque de servicios de entrada/salida, también es un grupo de servicios muy utilizado. La gestión del sistema de ficheros incluye las funcionalidades más conocidas también por los usuarios del intérprete de pedidos en relación con la gestión de archivos y directorios, como crear y eliminar entradas, cambiar de directorio o mostrar el directorio actual. También permite entablar las operaciones típicas con ficheros, como crear, eliminar, copiar o cambiar el nombre.

6) Protecciones. El sistema operativo tiene que combinar los atributos entre el proceso y el servicio que pide para determinar si este servicio se puede dar o no. Es una de las partes más utilizadas por los administradores de los sistemas, especialmente como llamadas del sistema, pero no lo es tanto por parte de los usuarios.

7) Funciones de tiempo. Todas las referencias temporales de los procesos provienen de llamadas al sistema operativo, como por ejemplo la fecha y la hora. También se pueden capturar datos sobre el tiempo de ejecución de un proceso, el tiempo de CPU u otras funciones estadísticas de esta clase.

1.2.2. Las bibliotecas

Los lenguajes de alto nivel nos permiten llevar a cabo operaciones que el lenguaje máquina no puede servir directamente, como por ejemplo:

- operaciones matemáticas (por ejemplo ln, sin, cos o sqrt)
- manipulación de cadenas de caracteres (como strcpy o strcat)
- operaciones de entrada/salida con formato (como readln, printf o fopen)

Los lenguajes de alto nivel permiten efectuar operaciones de este estilo directamente, pero para que se puedan ejecutar se tienen que añadir fragmentos de código nuevos que las resuelvan; son las bibliotecas de rutinas.

Las bibliotecas son un conjunto de módulos objeto organizados adecuadamente. Las suministra o bien el fabricante del sistema operativo o bien la institución que ha desarrollado el entorno a compilación. Las rutinas que contienen están catalogadas y se sabe perfectamente cómo se tienen que utilizar; tienen definidos los nombres de los procedimientos, los parámetros necesarios y la manera como se pasan, así como los posibles ficheros complementarios[4]. No es habitual disponer del código fuente de las bibliotecas.

[4]Un ejemplo de ficheros complementarios son los ficheros cabecera (por ejemplo `/usr/include/fcntl.h`) del lenguaje C.

Los procedimientos de las bibliotecas son muy genéricos y están pensados para que muchos programadores de diferentes tipos de aplicaciones puedan hacer uso de ellos. Dentro de las aplicaciones de sistema, generalmente se dispone de herramientas para construir nuevas bibliotecas. El programador puede dar formato de biblioteca a aquellos procedimientos más generales y más utilizados y comprobados por un programador, o un equipo de programadores, y utilizarlos así más fácilmente en diferentes aplicaciones.

Podemos agrupar las bibliotecas en las tres familias siguientes:

1) Bibliotecas del sistema: permiten invocar las llamadas al sistema de forma independiente del lenguaje máquina de nuestro sistema computador. De esta forma, el código fuente de las aplicaciones será portable entre sistemas computadores donde esté instalado el mismo sistema operativo.

2) Bibliotecas del lenguaje: son las más genéricas y normalmente son estandarizadas para cada lenguaje de programación de alto nivel, al menos en un conjunto de procedimientos. Las áreas más comunes son la gestión de ficheros y de dispositivos de entrada/salida, las funciones matemáticas y la manipulación de cadenas de caracteres.

3) Bibliotecas especializadas: son aquellas que, de manera opcional, se añaden al sistema con el fin de resolver problemas más específicos que no pueden solucionar las bibliotecas estándar. Un buen ejemplo son las bibliotecas especializadas en cálculos matemáticos (como cálculo matricial, estadístico o criptográfico) o las bibliotecas para visualización gráfica. También se suministran para la utilización de hardware muy específico, como es el caso de los entornos industriales.

1.2.3. El intérprete de comandos

Los usuarios de un sistema operativo suelen trabajar con el sistema utilizando un intérprete de comandos (*shell*), ya sea en **modo texto** o en **modo gráfico**.

Hay una serie de conceptos comunes a todos los intérpretes de comandos, como el establecimiento de una sesión de trabajo, la autenticación del usuario y la introducción de los comandos (ya sea utilizando el teclado o utilizando el ratón).

Este documento no describe el funcionamiento de ningún intérprete de comandos.

Ved también

Dentro de los recursos de vuestra aula disponéis de los enlaces necesarios para iniciaros en el intérprete de comandos típico de los sistemas Linux.

1.3. El sistema operativo desde el punto de vista del programador

La visión del usuario del sistema operativo está definida principalmente por las aplicaciones de sistema y, concretamente, por la interacción con el intérprete de pedidos. En cambio, el programador tiene un punto de vista muy diferente; allí donde el usuario ve las facilidades ofrecidas por el sistema operativo, el programador ve los recursos físicos y, mediante llamadas al sistema operativo, tiene que conseguir crear un programa que dé facilidades a un futuro usuario.

El programador dispone de las dos vías siguientes para utilizar los servicios del sistema operativo:

- Las llamadas al sistema hacen que en tiempo de ejecución se transfiera el control a un código que pertenece al núcleo del sistema operativo. Por lo tanto, el programador no debe implementar dentro del programa estos servicios, únicamente debe solicitarlos.

- Las bibliotecas de sistema, que son un conjunto de procedimientos de uso común que se pueden añadir a las aplicaciones mediante una simple referencia al procedimiento en cuestión. Eso hace que el código que necesitamos se enlace con el nuestro durante el proceso de compilación. En general, el código de las bibliotecas es de nivel más alto y es más independiente del hardware que se utiliza que el código de las llamadas al sistema.

Ved también

El anexo da un repaso al conjunto de pasos necesarios para generar un fichero ejecutable (compilación, enlazado o montaje) y ejecutarlo.

2. Los mecanismos de entrada al sistema operativo

En este apartado, trataremos los mecanismos para transferir la ejecución al sistema operativo desde un nivel más bajo (próximo al hardware) hasta un nivel más alto (más abstracto). Algunos de estos mecanismos responderán directamente a la voluntad del proceso en ejecución, pero otros serán ajenos.

2.1. Mecanismos directos (*traps*, excepciones e interrupciones)

Hay tres mecanismos básicos mediante los cuales se puede transferir la ejecución al sistema operativo:

- **Llamadas al sistema operativo** (también llamadas *system calls* o *traps*): las llamadas al sistema son provocadas por las aplicaciones de usuario cuando piden un servicio al sistema operativo. En un punto concreto del código de usuario se hace una transferencia explícita del control al sistema operativo mediante la ejecución de la instrucción del lenguaje máquina que permite hacerlo (en el caso de los procesadores Intel es la instrucción int). Desde este punto de vista, son acontecimientos síncronos. También pueden recibir el nombre de interrupciones de software. El sistema operativo define cuáles son los parámetros de entrada y de salida de cada llamada al sistema, así como la forma de pasar estos parámetros (por registros o por la pila, entre otros).

- **Interrupciones**: las interrupciones son provocadas por los dispositivos del hardware cuando quieren informar de algún cambio de estado (por ejemplo, la pulsación de una tecla o la finalización de una operación de entrada/salida). Normalmente, son éxitos asíncronos ajenos al proceso en ejecución. Típicamente reciben el nombre de interrupciones hardware.

- **Excepciones**: las excepciones se producen cuando el hardware detecta algún tipo de anomalía. Por ejemplo, si el procesador está ejecutando la instrucción de lenguaje máquina *dividir* y el denominador tiene el valor 0, el procesador genera una excepción porque esta operación no está definida. Las excepciones son rupturas de secuencia no previstas; de hecho son interrupciones, pero provocadas directamente por la ejecución del código mismo del usuario en curso.

A pesar de estas diferencias, el tratamiento que reciben estos acontecimientos es muy similar:

- **Salvar el estado del proceso de usuario en ejecución**: en el momento de producirse el acontecimiento que causa la llamada al sistema / interrupción / excepción, lo más probable es que haya algún proceso de usuario

en ejecución. Con el fin de poder ejecutar la rutina de atención del sistema operativo, hay que detener la ejecución de este proceso de usuario. Ahora bien, lo más probable es que, una vez ejecutada esta rutina, haya que reanudar la ejecución del proceso. Para poder hacerlo, hay que salvar el estado del proceso (contenido de los registros del procesador, por ejemplo) y "tomar una foto" del proceso, de forma que más adelante éste pueda continuar su ejecución como si nada hubiera pasado.

- **Determinar qué rutina del sistema operativo tiene que tratar este acontecimiento**: existen diversas alternativas para realizar esta determinación. Linux resuelve este problema en tiempo de ejecución (cada vez que se produce un *trap*/interrupción/excepción) mediante un vector de interrupciones que almacena en cada posición una posición de memoria, que es la posición donde se encuentra el comienzo de la rutina de atención asociada a aquella interrupción.
  Cada posible excepción e interrupción tiene asociada una posición fija de este vector. Cuando se produce una interrupción o una excepción, el hardware consulta la entrada correspondiente del vector de interrupciones y determina adónde se tiene que transferir el control.
  El caso de las llamadas al sistema es ligeramente diferente. Como el número de llamadas al sistema ofrecidas por un sistema operativo puede ser muy grande (337 en el caso de la versión 2.6.32 de Linux), se ha optado por un mecanismo híbrido hardware-software. Todas las llamadas al sistema tienen asociada la misma entrada del vector de interrupciones (la 0x80); la rutina de atención en el *trap* 0x80 es la que determina adónde hay que transferir el control en función de un registro del procesador que contiene el identificador de llamada al sistema que se quiere ejecutar.

- **Ejecutar la rutina de atención**: se transfiere el control a la rutina del sistema operativo asociada a este acontecimiento.
  - En el caso de las llamadas al sistema, esta rutina implementará el servicio pedido por el usuario (por ejemplo, leer del teclado o ejecutar un programa).

  - En el caso de las interrupciones, la rutina de atención actúa en consecuencia (por ejemplo, si se ha pulsado una tecla, guarda el carácter leído en una memoria intermedia o, si ha finalizado una lectura de disco, informa al proceso que lo estaba esperando).

  - En el caso de las excepciones, la rutina de atención intentará solucionar el problema. A partir de aquí no se puede generalizar, ya que a veces el sistema puede solucionar el problema y a veces no queda más alternativa que abortar la ejecución del proceso.

- **Continuar la ejecución de un proceso de usuario**: en algunos casos será el mismo que estaba en ejecución antes de producirse el acontecimiento,

mientras que en otros casos será otro proceso. Para hacerlo, hay que restaurar el estado (que previamente ha sido salvado) y continuar su ejecución.

Aunque estos mecanismos puedan recordar las rutinas convencionales (hay que salvar el estado, se produce una ruptura de la secuenciación implícita), existen unas diferencias muy importantes:

- El código de la rutina de atención en los *traps*/interrupciones/excepciones no forma parte de los ejecutables de usuario, sino que forma parte del núcleo del sistema operativo.

- La dirección de memoria donde se encuentra el código de la rutina de atención se determina en tiempo de ejecución (en el caso de las rutinas convencionales, la dirección de la rutina es un parámetro de la instrucción de lenguaje máquina que invoca una rutina).

- Otra diferencia tiene que ver con el modo de ejecución y la veremos más adelante en este módulo. Todas estas rutinas de atención forman parte del núcleo del sistema operativo y tienen acceso completo al hardware; en cambio, las rutinas escritas por el usuario no tendrán acceso a determinadas partes del hardware.

2.2. Mecanismos indirectos

2.2.1. Biblioteca del sistema

Desde el punto de vista del programador, el mecanismo de entrada al sistema mediante *traps* tiene un gran problema: no es portable porque depende del lenguaje máquina propio del procesador. Si queremos facilitar que el código fuente de los programas sea fácilmente portable entre arquitecturas diferentes donde esté instalado el mismo sistema operativo, hay que disponer de una forma independiente del hardware para poder realizar las llamadas al sistema operativo. La solución viene dada por la biblioteca del sistema. Las rutinas de esta biblioteca permiten expresar de forma estándar las llamadas al sistema.

Por ejemplo, la figura 2 muestra la interfaz (en lenguaje C) de la llamada al sistema write del sistema operativo Unix.

```
"ssize_t write(int fd, const void *buf, size_t count);"
```

Figura 2. Interfaz de la llamada al sistema write del sistema operativo Linux

La rutina write de la biblioteca del sistema Unix para la arquitectura IA-32 invoca la llamada al sistema tal como se hace en esta arquitectura (paso de parámetros a través de registros determinados, instrucción de lenguaje máquina int). La figura 3 (arriba) muestra el código ensamblador de esta rutina.

```
000001cd <write>:
     1cd:    push    %ebp
     1ce:    mov     %esp,%ebp
     1d0:    push    %ebx
     1d1:    sub     $0x4,%esp
     1d4:    mov     $0x4,%eax
     1d9:    mov     0x8(%ebp),%ebx
     1dc:    mov     0xc(%ebp),%ecx
     1df:    mov     0x10(%ebp),%edx
     1e2:    int     $0x80
     1e4:    mov     %eax,0xfffffff8(%ebp)
     1e7:    cmpl    $0xffffff82,0xfffffff8(%ebp)
     1eb:    jbe     1f4 &lt;write+0x27&gt;
     1ed:    movl    $0xffffffff,0xfffffff8(%ebp)
     1f4:    mov     0xfffffff8(%ebp),%eax
     1f7:    add     $0x4,%esp
     1fa:    pop     %ebx
     1fb:    leave
     1fc:    ret
```

```
__write:
0x120009fd0: addq zero, 0x4, v0
0x120009fd4: call_pal callsys
0x120009fd8: beq a3, 0x120009ff0
0x120009fdc: br gp, 0x120009fe0
0x120009fe0: ldah gp, 0(gp)
0x120009fe4: lda gp, 19472(gp)
0x120009fe8: lda at, 26144(gp)
0x120009fec: br zero, 0x120015210
0x120009ff0: ret zero, (ra), 1
```

Figura 3. Código de la rutina de biblioteca write de Unix en dos arquitecturas: IA-32 (arriba) y Alpha (abajo)

De esta forma, el programador puede invocar la llamada al sistema write como si fuera una rutina cualquiera del lenguaje C. Como el ejecutable se genera sobre la arquitectura IA-32, se utilizará la biblioteca de sistema correspondiente a la arquitectura IA-32.

La figura 3 (abajo) muestra el código ensamblador de la rutina `write` de la biblioteca del sistema Unix para la arquitectura Alpha. Como el ejecutable se genera sobre esta arquitectura, se utilizará la biblioteca de sistema de la

arquitectura Alpha. De esta forma, el programador no necesita conocer cómo se da la llamada al sistema en lenguaje máquina y puede invocarla de una forma portable.

Los otros dos mecanismos (interrupciones y excepciones) no están a disposición directa del usuario.

2.2.2. Bibliotecas del lenguaje

Algunos lenguajes de programación ofrecen servicios más próximos a las necesidades del programador y, por lo tanto, permiten que el programador sea más productivo.

En el Unix, por ejemplo, las llamadas de entrada/salida (read/write) trabajan con cadenas de bytes. Si queremos escribir un valor entero en base 10 por pantalla, antes es necesario convertir el valor entero a la cadena de dígitos decimales que representa el valor.

En cambio, el lenguaje C ofrece una biblioteca de entrada/salida estándar (la biblioteca stdio con rutinas como printf o scanf) que permite realizar entrada/salida con formato.

La figura 4 muestra un ejemplo de cómo escribir en el Unix un número entero por pantalla en base 10. Si utilizamos directamente llamadas al sistema (código de la izquierda), hay que transformar este valor numérico en la cadena de caracteres que representa su valor. Si utilizamos la rutina de biblioteca printf (código de la derecha), esta rutina realiza la conversión e invoca la llamada al sistema correspondiente.

```
char s[10];
int i=9;

s[9]='\n';
while (n > 0) {
       i--;
       s[i] = '0' + (n%10);
       n=n/10;
}
write(1, s[i], 10-i);
```

```
int n;

printf("%d\n", n);
```

Figura 4. Cómo escribir en Unix un número entero por pantalla en base 10: utilizando llamadas al sistema (izquierda) y utilizando rutinas de la biblioteca del lenguaje (derecha)

La biblioteca del lenguaje también ofrece rutinas de uso común que están implementadas sin utilizar llamadas al sistema, como, por ejemplo, rutinas trigonométricas, generadores de números pseudoaleatorios o manipulación de cadenas de caracteres (*strings*).

La figura 5 muestra un ejemplo en el que aparece una aplicación que utiliza rutinas de una biblioteca del lenguaje. Podemos ver los pasos necesarios hasta llegar al código del sistema operativo que implementa el servicio solicitado.

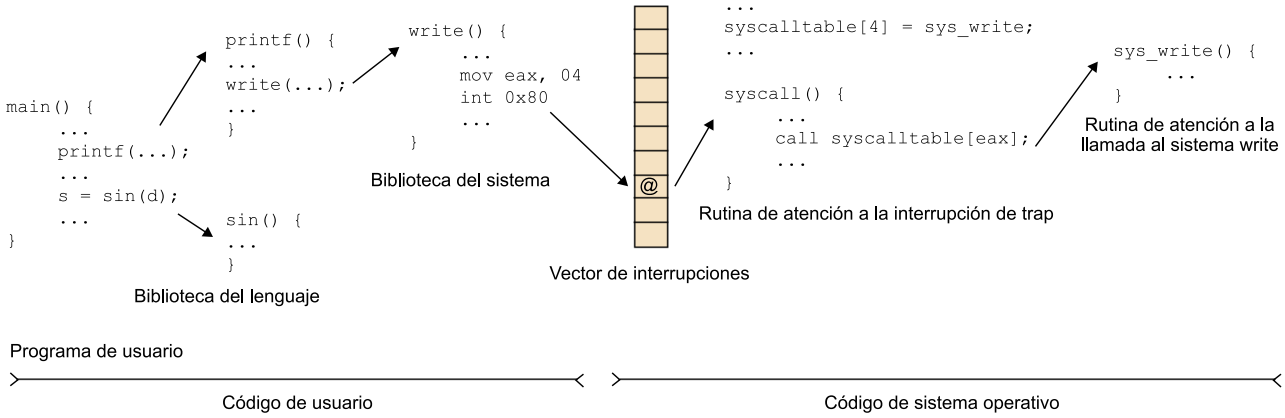

Figura 5. Entrada indirecta al código del sistema operativo utilizando una rutina de la biblioteca del lenguaje

2.2.3. Aplicaciones

Los usuarios no programadores quieren poder resolver sus necesidades sin tener que programar. El software de sistema les ofrece una serie de aplicaciones ya implementadas que resuelven sus necesidades más básicas. Estas aplicaciones, como no puede ser de otra forma, están implementadas utilizando las llamadas al sistema y las bibliotecas del lenguaje y del sistema.

Por ejemplo, el pedido cp del Unix (hace la copia de un fichero origen en un fichero de destino) está implementado utilizando una serie de llamadas al sistema: abrir el fichero de origen, abrir el fichero de destino, leer el contenido del fichero de origen, escribirlo en el fichero de destino y cerrar los ficheros involucrados.

La capa de aplicaciones de sistema ofrece una gran cantidad de aplicaciones que resuelven la gran mayoría de necesidades de los usuarios finales no programadores: crear/borrar/renombrar/copiar ficheros ordinarios y directorios, aplicaciones de ordenación de ficheros, consultar información sobre el sistema, detener/retomar/finalizar un proceso, entre otros.

3. Apoyo del hardware para implementar el sistema operativo

Hemos visto que el sistema operativo ofrece una máquina virtual a los usuarios más sencilla de usar que la máquina física. Ahora bien, ¿qué pasaría si un usuario insistiera en trabajar directamente con la máquina física ignorando el sistema operativo? Si el sistema operativo quiere garantizar la fiabilidad de los servicios que ofrece, no puede consentir esta situación. Un usuario que trabaje directamente con la máquina física puede provocar errores irrecuperables o puede acceder a recursos (como posiciones de memoria o sectores de disco) que no le pertenecen y poner en compromiso la confidencialidad y integridad de los datos de otros usuarios.

Ahora bien, al fin y al cabo, todo el código en ejecución en un procesador no es más que secuencias de instrucciones de lenguaje máquina. En principio, cualquier usuario podría escribir un programa que utilizara instrucciones del lenguaje máquina, que trabajen directamente con la máquina física e intentar ejecutarlo. Para poder neutralizar esta estrategia, el sistema operativo necesita un cierto apoyo por parte del hardware.

1) Dentro del repertorio de instrucciones del lenguaje máquina, la arquitectura del computador identifica un subconjunto de **instrucciones privilegiadas** que pueden poner en compromiso la estabilidad de la máquina o la confidencialidad de los datos, como por ejemplo la instrucción que permite detener el procesador, la que permite inhibir el tratamiento de las interrupciones o las que permiten leer/escribir los registros de control de los dispositivos de entrada/salida. Veremos que los programas de usuario no podrán utilizar directamente estas instrucciones privilegiadas. El resto de instrucciones del lenguaje máquina sí podrán ser utilizadas libremente por los programas de usuario.

2) El procesador ofrece dos (o más) modos de ejecución, que son **el modo de ejecución privilegiado y el modo no privilegiado** (usuario). En el modo privilegiado, el procesador podrá ejecutar cualquier instrucción de lenguaje máquina. En el modo usuario, el procesador únicamente podrá ejecutar instrucciones no privilegiadas; si intenta ejecutar una privilegiada, el procesador producirá una excepción.

3) El modo de ejecución usuario se utilizará para ejecutar el código de usuario y el modo de ejecución privilegiado se utilizará para ejecutar el código del sistema operativo. El procesador tendrá que controlar en cada momento en qué modo de ejecución se encuentra. Para hacerlo, sólo tendrá que ser consciente del momento en el que se producen los acontecimientos directos de entrada al sistema operativo (interrupciones, excepciones y llamadas al sistema) y del momento en el que se ejecuta la instrucción de lenguaje

máquina que indica el retorno desde una rutina de atención (en la arquitectura IA-32 es la instrucción iret). La figura 6 muestra el diagrama de estados correspondiente.

4) El vector de interrupciones (VI) determina qué rutinas dan servicio a las interrupciones/excepciones/llamadas al sistema. Obviamente, hay que garantizar que un proceso de usuario no pueda modificarlo. Si el VI se puede leer/modificar con instrucciones convencionales de acceso a la memoria, hay que garantizar que los procesos de usuario no puedan hacerlo. Es necesario que los procesadores dispongan de una unidad de gestión de la memoria (MMU, *Memory Management Unit*) que limite el rango o los rangos de direcciones de memoria que puede acceder cada proceso. Si un proceso intenta acceder fuera del rango o de los rangos que le corresponden, la MMU generará una excepción de acceso a la memoria inválido.

5) Con el fin de garantizar que el sistema operativo tome el control de la máquina periódicamente, es necesario un dispositivo temporizador que genere interrupciones periódicamente (por lo general, entre cien y mil veces por segundo).

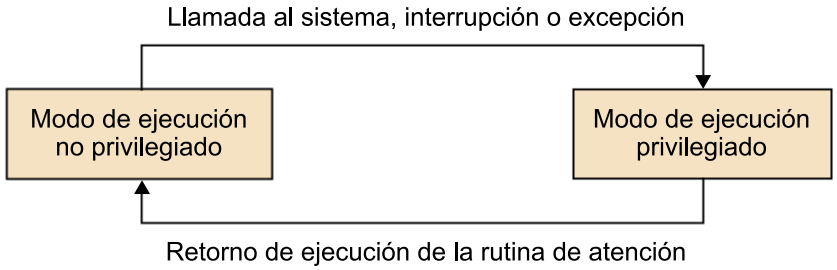

Figura 7. Diagrama que muestra las transiciones entre el modo de ejecución privilegiado y el no privilegiado.

Observaciones:

- El sistema operativo MS-DOS tenía graves problemas de estabilidad porque el procesador para el que se diseñó (el 8088/8086) tenía bastantes limitaciones (no disponía de varios modos de ejecución, no tenía instrucciones privilegiadas y no tenía MMU). Por lo tanto, cualquier programa en ejecución en la máquina tenía el control total de ésta.

- Gracias a la MMU, las instrucciones de lenguaje máquina de acceso a la memoria (load/store) no son privilegiadas. La MMU es quien determina si los accesos a la memoria que establece un proceso son válidos o no; si no lo son, la MMU generará una excepción que, en la mayoría de los casos, provocará que el sistema operativo aborte la ejecución del proceso.

Ved también

En el módulo 3, "La gestión de la memoria", trataremos la cuestión del acceso a la memoria con más detalle.

4. Introducción a la planificación del procesador

4.1. El concepto de proceso

La noción de proceso[5] es fundamental para entender el funcionamiento interno de los sistemas de computación modernos que son capaces de ejecutar muchas actividades de manera simultánea (sistemas operativos multiproceso). Sin embargo, no es sencillo ofrecer una definición exacta. A continuación, vamos a dar algunas de las más utilizadas:

(5)El término *proceso* aplicado al mundo de la computación fue utilizado por primera vez por el sistema operativo Multics en la década de 1960. Desde entonces, también se utiliza el término *tarea* como sinónimo.

- un programa en ejecución,
- el espíritu animado de un procedimiento,
- el centro de control de un procedimiento en ejecución,
- la entidad a la que se asignan los recursos de un ordenador,
- la instanciación de un programa.

Aunque, como hemos dicho, no hay ningún acuerdo sobre la definición de proceso, la expresión *un programa en ejecución* es una de las definiciones más aceptadas. Así, podemos formular las afirmaciones siguientes:

- Un **programa** es la descripción detallada para la resolución de un problema de manera general, en abstracto.

- Un **proceso** es la aplicación en concreto del programa a un caso particular y en un momento determinado con la asignación de unos recursos concretos. Un proceso es irrepetible.

De hecho, un mismo programa puede tener muchas instanciaciones (copias) que se ejecutan de manera concurrente (simultánea) en el mismo sistema informático. Por ejemplo, imaginemos que en un sistema multiusuario un conjunto de programadores están utilizando el mismo editor de texto.

Ved también

En el módulo 3 veremos si es necesario replicar el mismo código en la memoria tantas veces como usuarios hay para ejecutarlo.

Analogías de proceso

Podríamos decir que un programa es a un proceso como el fútbol es a un partido. El fútbol es un deporte en el que se han definido unas normas de funcionamiento, que resuelven todas las posibles incidencias y lo hacen de manera general. Cuando asignamos recursos a este concepto (como campo, entrenador y jugadores), instanciamos el concepto de fútbol y se inicia un nuevo proceso: el partido. Durante esa ejecución, el partido se desarrolla de una manera determinada y da unos resultados concretos, pero cuando finaliza deja de existir. El fútbol, en cambio, ha sido ajeno a esta ejecución y permite tantas otras instancias del juego como sean necesarias. La misma idea se puede aplicar a la descripción precisa de una interpretación musical (la partitura) y la ejecución correspondiente (el concierto) o a una receta de cocina y la preparación de un plato utilizando la receta.

Hay sistemas operativos que aceptan que se puedan definir diferentes hilos de ejecución concurrente (*threads*) dentro de un proceso.

Ved también

La definición de diferentes hilos de ejecución concurrente dentro de un proceso se tratará en el módulo didáctico de procesos.

4.2. Planificación del procesador

En un sistema operativo multiproceso existen diversos procesos que compiten para ejecutarse y hacer uso del procesador o de los procesadores disponibles en el ordenador. En este subapartado, asumimos que el ordenador dispone de un único procesador; en un momento dado se podrá estar ejecutando un único proceso y el resto de procesos tendrán que esperar para poder hacer uso del mismo.

El sistema operativo decide en cada momento qué proceso puede hacer uso del procesador; la parte del sistema operativo que lleva a cabo esta tarea es el **planificador del procesador**. El planificador asocia a cada proceso un estado que describe su grado de actividad.

Los estados básicos que contempla cualquier planificador del procesador son los siguientes:

- *Run* (ejecución): el proceso está utilizando el procesador.
- *Ready* (preparado): el proceso no está utilizando el procesador porque algún otro proceso lo está utilizando.
- *Blocked* (bloqueado): el proceso no está utilizando el procesador porque el proceso está esperando que finalice una petición que se ha hecho al sistema operativo (por ejemplo, una operación de entrada/salida o la sincronización con otro proceso). Este estado también puede recibir el nombre de *wait* (espera).

La figura 7 muestra un diagrama con estos estados, así como las transiciones entre ellos. A continuación, detallamos las causas que provocan estas transiciones:

- de *ready* a *run*: el planificador ha decidido que el proceso puede hacer uso del procesador.
- de *run* a *ready*: el planificador ha decidido que el proceso tiene que dejar de hacer uso del procesador.
- de *run* a *blocked*: el proceso ha invocado alguna llamada al sistema que no se ha podido realizar inmediatamente (por ejemplo, una lectura de te-

clado). Este tipo de llamadas al sistema reciben el nombre de *llamadas al sistema bloqueadoras*.

- de *blocked* a *ready*: se ha completado la ejecución de la llamada al sistema solicitado por el proceso, con lo cual el proceso está preparado para volver a utilizar el procesador.

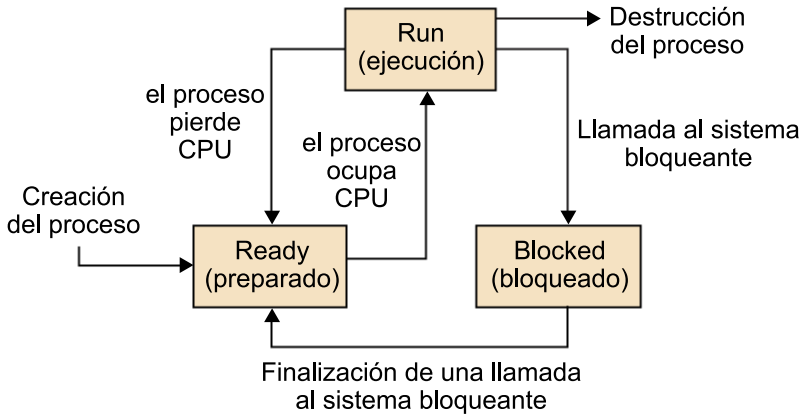

Figura 7. Versión simplificada del diagrama de estados de los procesos desde el punto de vista del planificador del procesador

Cuando el proceso que se encuentra en el estado *run* deja de utilizar el procesador (ya sea voluntariamente porque ha invocado una llamada al sistema bloqueante o bien involuntariamente porque el planificador del procesador ha decidido que otro proceso haga uso del procesador), se produce un cambio de contexto. El procesador detiene la ejecución de un proceso y pasa a ejecutar otro proceso. La rutina del sistema operativo que implementa el cambio de contexto tiene que guardar toda la información necesaria para, más adelante, poder reanudar la ejecución del proceso. Básicamente, esta información está formada por los registros del procesador.

El planificador del procesador es el responsable de garantizar que los procesos en ejecución en el sistema utilicen el procesador de forma equitativa. No es objetivo de esta asignatura ver los diferentes criterios que puede utilizar un planificador para desempeñar esta tarea.

Los diagramas de estados de los planificadores del procesador de los sistemas operativos actuales tienen más estados y transiciones que los que hemos presentado en la figura 7. Estos estados adicionales pueden mostrar si el proceso está ejecutando código en modo usuario o en modo sistema o el motivo que ha provocado el bloqueo del proceso, entre otros.

Resumen

En este módulo didáctico, hemos visto el sistema operativo como una máquina virtual que esconde a los usuarios la complejidad de la gestión de los recursos de hardware y software que componen un sistema computador. Hemos presentado las diferentes capas de software que podemos encontrar en un sistema informático y las hemos analizado desde las más próximas al hardware (rutinas de atención en interrupciones, excepciones y llamadas al sistema) hasta las más lejanas (aplicaciones de sistema), pasando por las bibliotecas de rutinas (del sistema y del lenguaje). Hemos visto que el código del sistema operativo responde a acontecimientos (interrupciones, excepciones y llamadas al sistema). Para complementar el tema, también se han listado los requisitos que tiene que cumplir el hardware para poder instalar un sistema operativo multiusuario que proteja a cada usuario del resto de usuarios. Finalmente, hemos hecho una breve introducción a la planificación del procesador.

Actividades

1. Utilizando alguno de los recursos disponibles en el aula, iniciad una sesión de trabajo en el intérprete de comandos de Linux e intentad hacer las siguientes pruebas:

a) Utilizad los pedidos básicos de gestión de ficheros y directorios (more, cat, cp, mv, rm, mkdir, cd, rmdir).

b) Utilizad el pedido ps para ver los procesos activos asociados a la sesión y todos los procesos en ejecución en el sistema.

c) Comprobad el funcionamiento de los filtros utilizando el metacarácter |.

d) Ejecutad pedidos en modo primer plano (*foreground*) y en modo segundo plano (*background*).

e) Generad el ejecutable de algún programa sencillo escrito en C (por ejemplo, un "Hello world"). Tendréis que usar las utilidades que hay en Linux para editar un programa escrito en C, compilarlo y montarlo.

2. Las versiones actuales de Linux basadas en la arquitectura IA-32 implementan las llamadas al sistema utilizando la instrucción sysenter en lugar de la instrucción int. Investigad qué ventajas tiene la utilización de sysenter respecto de la utilización de int.

Ejercicios de autoevaluación

1. ¿Qué diferencias hay entre invocar una rutina de un programa e invocar una llamada al sistema?

2. En este módulo, hemos mencionado algunos escenarios en los que se genera una excepción. ¿Cuáles son?

3. Indicad cuáles son las diferencias más relevantes entre los modos de ejecución del procesador (modo privilegiado y modo no privilegiado).

4. ¿En qué modo de ejecución se encuentra el procesador cuando...

a) un usuario ordinario ejecuta código propio de una aplicación de sistema (como el intérprete de comandos)?
b) un usuario privilegiado (el administrador de la máquina) ejecuta código propio de una aplicación de sistema?

5. ¿Cuáles de las siguientes instrucciones del lenguaje máquina son privilegiadas?

a) Leer el registro de control del disco.
b) Acceder a la memoria.
c) Sumar dos registros de propósito general del procesador.
d) Inhibir la atención en las interrupciones.
e) Hacer que el procesador pase a un modo de bajo consumo energético.

6. Desde el punto de vista del planificador del procesador, ¿qué diferencia existe entre que un proceso implemente una sincronización utilizando una espera activa o una sincronización utilizando una operación bloqueante?

7. ¿Qué se necesita para convertir un programa en proceso?

8. ¿Por qué motivos el MS-DOS era un sistema operativo inestable?

9. Desde el punto de vista del planificador del procesador, ¿por qué es necesaria la existencia de un dispositivo temporizador que genere interrupciones periódicamente?

10. Una vez leído el anexo de este módulo, contestad las preguntas:

a) ¿Podéis explicar las funciones del montador?
b) ¿Con qué otras herramientas del sistema tiene relación?
c) Imaginad que habéis escrito un único fichero de programa fuente, ¿por qué lo tenéis que montar?

Solucionario

Ejercicios de autoevaluación

1. Invocar una llamada al sistema provoca un cambio de modo de ejecución; la determinación de la posición de memoria donde se encuentra la rutina se hace en tiempo de ejecución y el código de la rutina no forma parte del fichero ejecutable.

2. En este módulo hemos visto: la división por cero, el intento de ejecutar una instrucción privilegiada del lenguaje máquina en modo de ejecución no privilegiado y un acceso a la memoria inválido.

3. En modo privilegiado, el procesador puede ejecutar cualquier instrucción del lenguaje máquina. En modo no privilegiado, únicamente se pueden ejecutar instrucciones no privilegiadas; si se intenta ejecutar una instrucción privilegiada en este modo de ejecución, se producirá una excepción.

4. En ambos casos se encuentra en modo de ejecución no privilegiado. El procesador se encuentra en modo de ejecución privilegiado únicamente cuando ejecuta código de atención a interrupciones, excepciones y llamadas al sistema. Las aplicaciones de sistema, independientemente de qué usuario las ejecute, siempre se ejecutan en modo de ejecución no privilegiado.

5. El acceso a la memoria y la suma de registros de propósito general son no privilegiadas; las otras son privilegiadas.

6. En el primer caso, el proceso está consumiendo CPU porque hace la transición entre los estados *run* y *ready*. En el segundo caso, el proceso no está consumiendo CPU porque durante todo el tiempo de espera tendrá asignado el estado Blocked.

7. Para que un programa genere un proceso nuevo el SO debe darle "vida"; le ha de asignar recursos, lo tiene que copiar en la memoria y tiene que hacer que el procesador inicie la ejecución.

8. Porque se diseñó para un procesador sin diversos modos de ejecución, instrucciones privilegiadas ni MMU.

9. Para evitar que un proceso de usuario de cálculo intensivo (que efectúa muy pocas llamadas al sistema) pueda monopolizar la CPU. El temporizador da la oportunidad al planificador del procesador de entrar en ejecución, con lo cual puede decidir forzar un cambio de contexto con otro proceso.

10.a) El montador combina diferentes módulos objeto y las bibliotecas para construir el programa ejecutable. El trabajo más importante que lleva a cabo es resolver las referencias entre las diferentes partes y establecer la correspondencia entre las direcciones de las diferentes variables y los procedimientos.

b) El montador tiene relación con las bibliotecas y con los módulos objeto generados durante la fase de compilación.

c) Siempre se tiene que utilizar el montador porque siempre se tienen que añadir las cabeceras para el cargador y las rutinas de las bibliotecas (es difícil encontrar una aplicación sin ninguna rutina de entrada/salida de las bibliotecas del sistema).

Glosario

biblioteca *f* Conjunto de rutinas de uso común ya implementados, disponibles para todos los usuarios. Este conjunto de rutinas codificadas como módulos objeto está organizado en ficheros que contienen el código con un índice que permite añadir este código al del programador. Este paso se da en la fase de montaje del fichero ejecutable.

cargador *m* Herramienta que permite cargar un programa en la memoria a fin de que pueda ser ejecutado; para lograrlo es necesario obtener memoria, copiar el ejecutable y darle control.

compilador *m* Herramienta que permite transformar un programa escrito en lenguaje de alto nivel en un programa escrito en un lenguaje conocido por la máquina, que permitirá que lo pueda ejecutar.

depurador *m* Herramienta que permite controlar la ejecución de un programa paso a paso analizando el contenido de las variables; de esta manera, nos facilita la detección de los errores y su corrección.

excepción *f* Transferencia de control al sistema operativo provocada por la ejecución de una instrucción de usuario no prevista como, por ejemplo, un fallo de página de memoria o una división por cero.

intérprete de comandos *m* Interfaz por medio de la cual el usuario interacciona con el sistema operativo. El intérprete de comandos se basa en el diálogo entre el usuario y el sistema; el sistema declara su disponibilidad presentando el indicador y el usuario pide servicios introduciendo pedidos. La mayoría de los intérpretes de pedidos admiten la programación de programas pequeños o *scripts*.
sin intérprete de órdenes
en *shell*

interrupción *f* Transferencia de control al núcleo del sistema operativo. Típicamente es provocada de manera asíncrona por los dispositivos del sistema por medio del hardware (interrupciones hardware).

librería *f* Nombre que también reciben las bibliotecas a causa de una traducción incorrecta del término inglés de *library*, un falso amigo del inglés, al catalán o al castellano.

llamada al sistema *f* Herramienta que permite al usuario utilizar unos servicios que se tienen que ejecutar dentro del núcleo del sistema.

montador *m* Herramienta que se encarga de agrupar los módulos y las bibliotecas necesarios para construir un programa ejecutable.

proceso (tarea) *m* Programa en ejecución.

programa *m* Descripción detallada en forma de instrucciones que permite resolver un problema determinado. Los programas tienen que estar escritos en ficheros de texto y tienen que seguir unas normas sintácticas determinadas por un lenguaje de programación.

programa fuente *m* Fichero que contiene las instrucciones del programa escritas en un lenguaje informático.

programa objeto *m* Fichero que contiene un módulo de un programa escrito en lenguaje máquina y la información necesaria para ser montado con otros programas.

trap *m* Mecanismo de hardware que nos permite establecer llamadas explícitas a los servicios del sistema operativo.

Bibliografía

Silberschatz, A.; Galvin; Gagne, G. (2008). *Operating Systems Concepts* (8.ª ed.). John Wiley & Sons.

Tanembaum, A. (2009). *Modern Operating Systems*. Prentice-Hall.

Documentación disponible en los recursos del aula sobre el intérprete de comandos de Linux.

Anexo

A) Las fases de ejecución de un programa

AA) El lenguaje informático

Los programadores de aplicaciones escriben sus programas en ficheros de texto que siguen las normas de un lenguaje informático determinado. Estas normas determinan la gramática del lenguaje de programación. Eso es debido al hecho de que se intenta utilizar un lenguaje tan cercano como sea posible al lenguaje natural con el fin de describir la solución del problema. Para las personas de habla inglesa los lenguajes de programación son bastante naturales, dado que la mayoría están escritos en su lengua. El fichero generado es de texto y, por lo tanto, no se puede ejecutar de manera inmediata.

En el momento de resolver un problema determinado, los programadores de aplicaciones informáticas tienen que escribir las instrucciones o las sentencias de un programa siguiendo unas normas. El formato que utilizan se llama **lenguaje informático**.

La figura 8 presenta un ejemplo de uso de lenguaje informático. El programa que se representa efectúa la multiplicación de dos variables a y b y deja el resultado a c. Supongamos que las variables a y b están inicializadas en los valores correspondientes al cálculo deseado.

```
"c = 0;
for (i = 0; i &lt; b; i++)
     c = c + a;"
```

Figura 8. Programa de ejemplo que calcula el producto de los enteros a y b (notad que el programa presentado es únicamente un ejemplo y no contiene todo el código necesario).

Un **lenguaje de programación o informático** es una convención de sentencias y estructuras de datos que, mediante un proceso de traducción o interpretación, se puede convertir en lenguaje máquina y se puede ejecutar. Con los lenguajes de programación se pueden escribir algoritmos.

La idea de los lenguajes de programación es que los programas sean independientes de la arquitectura física del ordenador y del sistema operativo en el que se ejecuten finalmente, es decir, que sean programas **portables a otros sistemas** y que sean útiles sin tener que introducir cambios. Organismos como el American National Standards Institute (ANSI) y la International Standards Organization (ISO) se ocupan de la normalización de los diferentes lenguajes.

Así pues, el sistema operativo tiene que proporcionar las herramientas necesarias para llevar a cabo todo el proceso de creación de una aplicación informática que se pueda ejecutar en el sistema del mismo sistema operativo.

Clasificamos los diferentes tipos de lenguajes de programación de la manera siguiente:

- **Lenguaje de alto nivel**. Cuando las instrucciones se escriben en un formato de texto muy cercano al lenguaje natural, aunque con unas reglas mucho más estrictas, decimos que estamos utilizando un lenguaje de alto nivel.
  El ejemplo anterior está escrito en un lenguaje de alto nivel. La máquina únicamente puede ejecutar el lenguaje máquina propio y, por lo tanto, tiene que haber un proceso que transforme el programa anterior a un formato conocido por la máquina. Este proceso se llama, de manera genérica, **compilación**.

- **Lenguaje ensamblador**. Para simplificar el uso directo del lenguaje máquina se han diseñado lenguajes en los que cada instrucción máquina corresponde a una instrucción en un lenguaje mnemotécnico más inteligible. Uno de estos lenguajes es el lenguaje ensamblador, que permite escribir un código muy eficiente por el hecho de que se pueden escribir las instrucciones nativas del procesador controlando toda la ejecución a un nivel muy bajo. Sin embargo, hay que decir que el lenguaje máquina es cada día más sofisticado y que es prácticamente inviable programarlo a mano, de manera que esta tarea tan complicada quedará reservada a los compiladores y a los optimizadores de código.

- **Código máquina**. Finalmente, las únicas instrucciones que reconoce un procesador son las que están codificadas en su código particular. Este código es binario, en el sentido de que se interpretan directamente las secuencias de ceros y unos. Se puede ejecutar de manera directa y recibe el nombre de código máquina. Normalmente, una sentencia de lenguaje de alto nivel genera diversas instrucciones de código máquina.

En la figura 9, vemos cómo sería el aspecto del programa del ejemplo anterior escrito en lenguaje de alto nivel, en lenguaje ensamblador y en código máquina. A la hora de codificar las instrucciones CALL y RET se utiliza el mismo código que para la instrucción BEQ y se utiliza el campo de origen para distinguir de qué instrucción se trata: 0 para BEQ, 1 para CALL y 2 para RET.

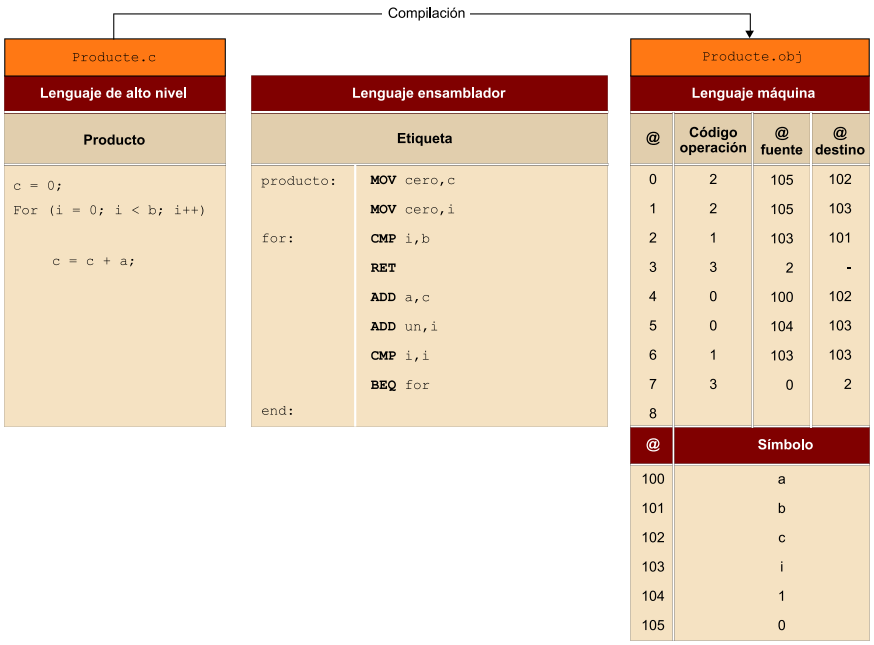

Figura 9. Tres versiones de una rutina de ejemplo en lenguaje de alto nivel, en lenguaje ensamblador y en lenguaje máquina (este ejemplo se basa en la idea de una máquina sencilla con un juego de instrucciones muy reducido, por ejemplo, no dispone de la instrucción producto).

Hemos representado el código máquina en notación decimal para cada campo. El código de operación sólo ocupa 2 bits y los códigos de la variable de origen y de destino ocupan 7 bits cada uno. Siempre dan un mensaje directo a la posición de memoria del operando, por lo que los valores constantes cero y uno se tienen que registrar en una posición de memoria.

La máquina no dispone de una operación de salto incondicional. Con el fin de simularla (ir a la etiqueta for:), hacemos que el resultado de la comparación previa sea siempre cierto, cosa que conseguimos comparando cualquier posición de memoria con ella misma (en este ejemplo, la variable i).

Ved también

Podéis ver el valor de las posiciones de memoria de cada variable en la tabla de símbolos que hemos definido en este anexo.

AB) Las herramientas para la creación de una aplicación informática

Dentro de los programas de utilidad del sistema operativo, hay un amplio abanico de catalogados como programas de apoyo a los lenguajes de programación, entre los cuales podemos encontrar las herramientas siguientes:

- **Editores de texto**: forman parte de las utilidades del sistema que permiten la edición de ficheros que contienen información en formato de texto. Estos ficheros pueden ser de mensajes, de configuración del sistema o de pequeños programas que utilizan el intérprete de comandos del sistema operativo (scripts o *shellscripts*), pero también pueden ser ficheros de código de alto nivel de una aplicación informática. Recordemos que los programas escritos en alto nivel no son sino ficheros de texto escritos con las normas de un lenguaje de alto nivel.

- **Compiladores-ensambladores**: son programas que transforman los programas editados en formato de texto a un formato que sea ejecutable por la máquina. Los ficheros resultantes se llaman ficheros ejecutables y su

contenido está representado en el formato binario, que el hardware puede ejecutar de manera directa. Tanto los programas fuente de alto nivel como los editados en ensamblador están escritos en formato de texto. No es habitual establecer la transformación de programas de alto nivel a ejecutables y que la transformación se pueda hacer en un solo paso. Más adelante detallaremos este proceso.

- **Bibliotecas**: para ayudar a crear programas, el sistema operativo permite la gestión de bibliotecas. Éstas permiten extender las funciones que se pueden utilizar de manera directa y, así, ahorran al programador el trabajo pesado de programar muchos procedimientos de entrada/salida o de gestión de ficheros, que se repiten para cualquier tipo de aplicación. Utilizando las bibliotecas aprovechamos un conjunto de rutinas que ya está muy depurado y experimentado.

- **Montadores (enlazadores o *linkers*)**: son aplicaciones que agrupan diferentes módulos objeto y las bibliotecas del sistema para obtener un programa ejecutable único. Su tarea principal es resolver las referencias cruzadas entre los diferentes elementos que no han sido resueltas en las primeras fases de compilación. Aunque a veces se encuentran como aplicaciones separadas de los sistemas operativos, normalmente se ejecutan de manera automática en la fase final de los entornos integrados de compilación.

- **Depuradores**: están formados por una serie de aplicaciones que permiten la ejecución controlada de un programa con el fin de resolver posibles errores. Para hacer viable la utilización de un depurador es necesario realizar las fases previas en esta modalidad. El compilador y el montador tienen que preparar el código resultante para este fin.

- **Ejecutadores-cargadores**: son el último paso del proceso y permiten la ejecución de los ficheros ejecutables resultantes de los pasos anteriores después de asignarles recursos del sistema. En los sistemas operativos antiguos esta orden era explícita, mientras que actualmente, al indicar el nombre de un fichero ejecutable, se interpreta que se tiene que iniciar su ejecución.

AC) El proceso de creación de un programa ejecutable

Utilizando las herramientas que hemos descrito podemos crear y ejecutar una aplicación. Para poder lograrlo, tenemos que pasar por un conjunto de fases que desarrollamos acto seguido.

ACA) La edición

En esta fase, el programador tiene que transcribir todas las ideas para solucionar los problemas en un fichero de texto llamado **fichero fuente**. Las instrucciones del programa se tienen que escribir siguiendo las normas estrictas del lenguaje de programación que se utilice.

La herramienta que se utiliza es cualquier procesador de texto que, de hecho, puede ser muy sencillo, ya que no se necesitan grandes prestaciones en cuanto al formato del texto. En cambio, es conveniente que pueda gestionar más de un fichero simultáneamente o que disponga de buenas herramientas de búsqueda y de manipulación del texto. Hay editores específicos para algunos lenguajes de programación que pueden resolver ciertos problemas sintácticos del código o que pueden presentar las partes de un programa en diferentes colores.

Ejemplo

Se pueden utilizar los recursos del editor de texto para diferenciar las palabras reservadas, los nombres de las variables o los nombres de los procedimientos, entre otros.

ACB) La compilación (ensamblaje)

Con el proceso de traducción (compilación) de un programa escrito en lenguaje de alto nivel, llamado programa fuente, obtenemos uno equivalente en lenguaje máquina que llamamos **programa objeto**. Si el programa fuente es el ensamblador, esta fase también se llama **ensamblar** y, en este caso, la traducción es más sencilla, dado que, como hemos dicho, a cada instrucción en ensamblador le corresponde una instrucción máquina. A partir de esta fase, no hay ninguna diferencia entre los objetos que provienen de un lenguaje de alto nivel y los que son de formato ensamblador.

Nota

Este programa equivalente tendrá la misma información, pero en un formato diferente.

La estructura de un programa objeto ya contiene los códigos correspondientes a las instrucciones máquina, pero el programa objeto no puede resolver todas las direcciones que incluye a causa de las dos razones siguientes:

- En primer lugar, porque los programas de una aplicación habitualmente están escritos en diferentes ficheros o módulos. Así, si tenemos que hacer referencia a una variable o a un procedimiento que está escrito en otro módulo, no la podemos resolver en este paso.

- En segundo lugar, porque las rutinas de uso muy común, como los cálculos matemáticos más o menos complejos, el control de determinadas operaciones de entrada/salida con ficheros o dispositivos y otros procedimientos muy genéricos ya están generalmente programadas, compiladas e incluidas en un conjunto de ficheros llamados bibliotecas.

Las **bibliotecas** son un conjunto de módulos objeto organizados adecuadamente. El mismo fabricante del software de apoyo a los lenguajes de programación suele facilitar este fichero. Gracias a las bibliotecas, el programador se ahorra mucho trabajo y se reducen considerablemente el tamaño del pro-

grama y el tiempo de programación. Normalmente, el fabricante no facilita el programa fuente de las bibliotecas, pero sí proporciona las instrucciones y los parámetros necesarios para utilizarlas.

Tanto los módulos objeto como las bibliotecas indican las direcciones de manera relativa al inicio del módulo. Después de esta primera fase de compilación, disponemos de un conjunto de módulos objeto con referencias de menaje relativas al módulo y también otras referencias externas sin resolver. En esta fase se crean tres estructuras:

- **La tabla de símbolos**, que indica la ubicación de cada símbolo definido en un módulo objeto (variable, procedimiento o constante).
- **La tabla de referencias externas**, que indica qué instrucciones hacen referencia a símbolos no definidos dentro de un módulo objeto.
- **La tabla de reubicación**, que indica qué referencias a símbolos se tendrán que modificar en el proceso de montaje. Sin embargo, no es objeto de esta asignatura dar los detalles de cómo se tratan estas tablas ni de la manera como se utilizan.

Ejemplo de compilación

Consideramos que hay dos módulos, producte.c y principal.c. En principal.c (figura 10), encontramos un bucle que imprime los cuadrados de los n primeros números. En el ejemplo hemos tomado el valor $n = 10$. Este programa hace una serie de llamadas a un segundo módulo en el que hemos escrito el código para el producto. Recordad que sólo es un ejemplo didáctico en el que no lo hemos resuelto todo. Primeramente podemos ver, de manera muy simplificada, cómo es la compilación de producte.c. Presentamos el código ensamblador sólo para intuir el significado del código máquina. Las constantes 0 y 1 se han cargado en posiciones de memoria, ya que la máquina dispone únicamente de direccionamiento directo.

Es importante ver que ahora hay una serie de referencias externas no resueltas, como las direcciones de las variables a y b y las de los procedimientos producte y printf.

```
n = 10;
for (i = 0; i &#8249; n; i++){
  a = i;
  b = i;
  producte();
  printf(_i = %d quadrat =...
}
```

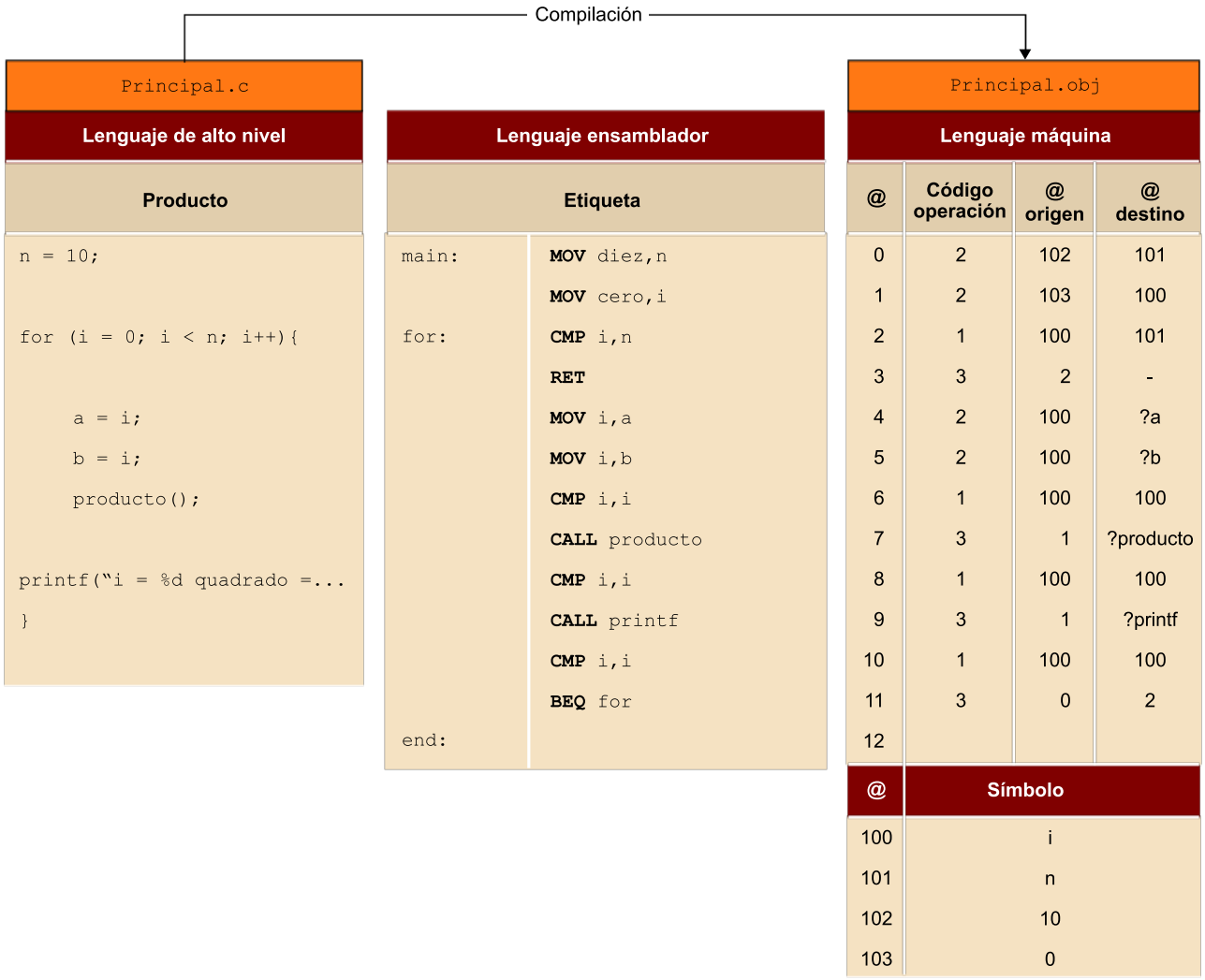

Figura 10. Código en alto nivel, ensamblador y lenguaje máquina del programa principal.c

ACC) El montaje (enlazado o linkaje)

El montaje es el proceso encargado de coger todos los módulos correspondientes a un programa, y también las bibliotecas si es necesario, y agruparlos para construir el programa ejecutable. Los programas que llevan a cabo estas funciones se llaman montadores.

De hecho, el trabajo principal del enlazador es resolver las direcciones externas entre los módulos y también con la biblioteca. El resultado final del enlazador es el programa ejecutable definitivo con un espacio lógico de direcciones completamente construido.

La **cabecera de un fichero ejecutable** contiene información relativa a la manera como se tiene que cargar en la memoria, como el tamaño de la pila y de los datos no inicializados o dinámicas, el número de segmentos (si los hay) o la especificación de si el código puede ser compartido con otros procesos, entre otros. También indica si se genera código para su depuración y, en ese caso, adjunta información con el nombre de las variables, de los procedimientos y de

las direcciones que sean necesarias. Es habitual que la cabecera contenga una marca, que es la **firma del fichero**. Con este identificador, el sistema operativo sabe cuándo un fichero es realmente ejecutable, siempre que la versión del sistema operativo sea correcta o el código máquina generado sea el apropiado.

El enlazado de los módulos del ejemplo propuesto es el que podéis ver en el gráfico de la figura 11.

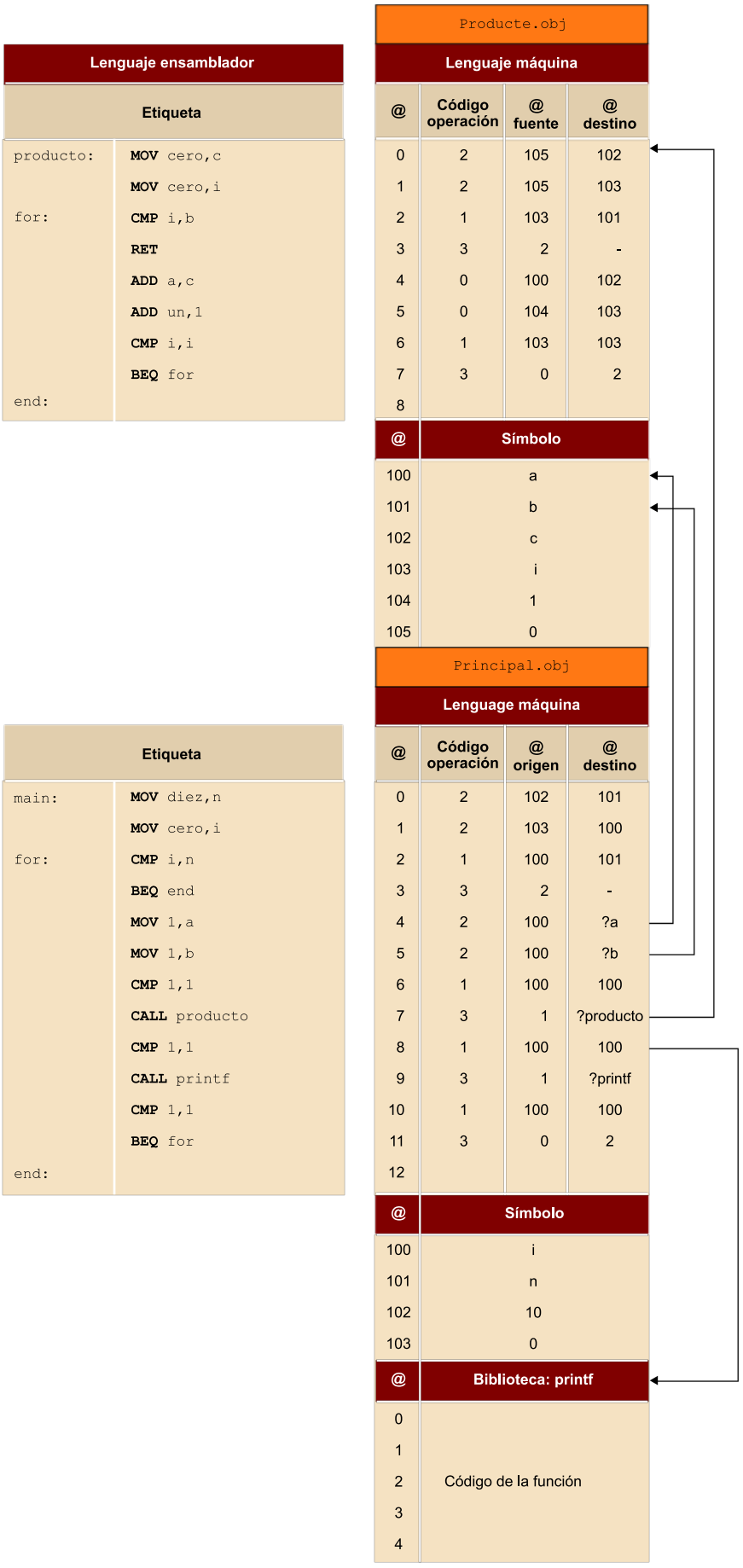

Figura 11. Enlazado de los módulos de ejemplo

Podemos ver que a la hora de generar el fichero ejecutable surgen diferentes problemas:

- Las direcciones de las tres partes que se tienen que enlazar empiezan por 0.
- Hay dos símbolos locales con el mismo nombre: i.
- Existen variables y procedimientos todavía sin montar.
- Las direcciones de los saltos incondicionales (BEQ) pueden cambiar.

Para solucionar estos problemas, el fichero ejecutable podría ser parecido a lo que presentamos en la figura 12.

Como podemos ver, se han resuelto las referencias externas. El compilador no puede simplificar la doble definición de la constante del cero y la mantiene por separado. El orden de los módulos es irrelevante, de manera que podríamos crear un ejecutable equivalente alterando el orden de los módulos. También hay que subrayar que el código de la biblioteca se ha gestionado como un módulo más y que se ha añadido el código de la rutina de biblioteca printf. Por otra parte, si alteramos un solo módulo objeto, claro está que el resto de objetos son perfectamente válidos para su montaje.

Lenguaje ensamblador	
Etiqueta	
producto:	**MOV** cero1,c
	MOV cero1,i1
for1:	**CMP** i1,b
	RET
	ADD a,c
	ADD un,i1
	CMP i1,i1
	BEQ for 1
main:	**MOV** diez,n
	MOV cero2,i2
for2:	**CMP** i2,n
	RET
	MOV i2,a
	MOV i2,b
	CMP i2,i2
	CALL producto
	CMP i2,i2
	CALL printf
	CMP i2,i2
	BEQ for2
end2:	

Executable.exe

Punto de entrada: @8 Versión del sistema, otros datos			
@	**Código operación**	**@ fuente**	**@ destino**
0	2	105	102
1	2	105	103
2	1	103	101
3	3	2	-
4	0	100	102
5	0	104	103
6	1	103	103
7	3	0	2
8	2	108	102
9	2	109	106
10	1	106	107
11	3	2	-
12	2	106	100
13	2	106	101
14	1	106	106
15	3	1	0
16	1	106	106
17	3	1	20
18	1	106	106
19	3	0	10
20		Printf	
21			
22		Código de la función	
23			
24		(RET)	

@	Símbolos
100	a
101	b
102	c
103	i1
104	1
105	0
106	i2
107	n
108	10
109	0

Cabecera

Imagen del código ejecutable

Figura 12. Fichero ejecutable correspondiente al programa de ejemplo

ACD) La carga

La tarea principal del cargador es buscar los recursos que pide el programa ejecutable. Entre estos recursos está la memoria. El **cargador** tiene que buscar memoria libre, copiar el código y también asignar espacio para datos y pila.

Todo este proceso se hace adaptando las direcciones lógicas de los programas ejecutables a las direcciones físicas de la memoria real del sistema. Finalmente, el cargador transfiere el control a la primera instrucción del programa y marca el proceso como preparado.

Cuando muchos usuarios llaman al mismo programa, si éste permite que se utilice en modalidad compartida, el cargador mirará si hay alguna copia en la memoria del programa. Si la respuesta es afirmativa, hará que el nuevo proceso tenga todo su entorno propio excepto la parte de memoria donde reside el código, que será compartida con otros procesos. Recordemos el ejemplo de los programadores que utilizan al mismo tiempo el editor.

ACE) La ejecución (depuración)

Mientras se lleva a cabo el proceso de desarrollo de un programa o de una aplicación informática, suele ser muy interesante seguir la ejecución paso a paso, cosa que es factible gracias a los programas depuradores (*debuggers*).

Los **programas depuradores** tienen como única misión controlar la ejecución de un programa y permiten que el programador pueda ver una ejecución paso a paso (instrucción por instrucción) o por intervalos de programa visualizando constantemente el entorno que se va produciendo. El programador fija los puntos donde se tiene que detener la ejecución y en cada parada puede comprobar si el entorno es el correcto e, incluso, puede modificarlo.

Así, se pueden ver y corregir errores que difícilmente se encontrarían de otra manera y se obtiene un programa final más eficiente (figura 13).

El hecho de partir el programa fuente en diferentes ficheros presenta varias ventajas tanto para el programador como para el sistema:

- Por una parte, porque una aplicación informática puede tener decenas de miles de líneas de código y editar un fichero de estas dimensiones puede resultar poco práctico.

- El proceso de compilación también sale beneficiado porque en la primera fase sólo tiene que traducir los módulos de código fuente que se hayan modificado. En la fase final o de enlace, sí tiene que actuar en todos los módulos objeto, tanto los que se han cambiado como los antiguos.

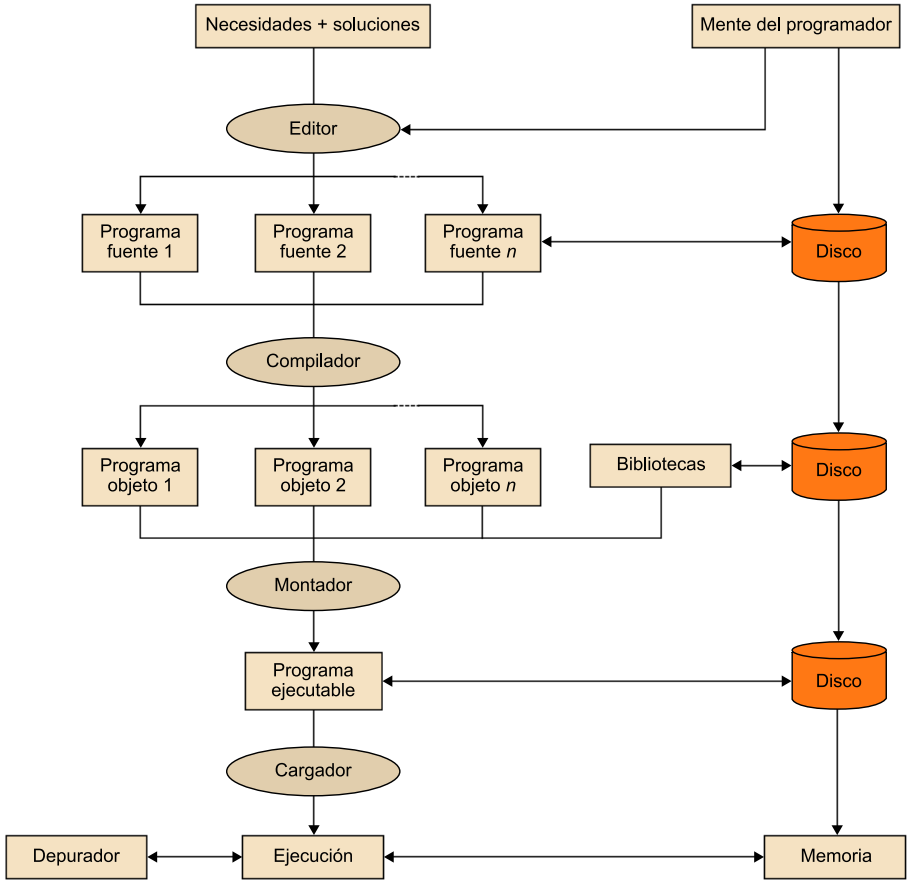

Figura 13. Etapas de la creación de un programa ejecutable

ACF) Interpretación frente a compilación

Los intérpretes intentan unificar todo el proceso de creación de un programa (figura 15). A medida que se va leyendo cada instrucción del lenguaje de alto nivel, se transforma en el conjunto correspondiente de instrucciones máquina, que se ejecuta inmediatamente. El interpretador controla directamente todas las fases de la creación y la ejecución del programa. Es un paso necesario para la ejecución, ya que no se genera ningún otro fichero y siempre se trabaja con el fichero fuente.

La interpretación de programas fue el primer proceso que apareció, dado que los ordenadores eran poco potentes y el proceso de compilación demasiado costoso. Era demasiado lento para aplicarlo continuamente durante el proceso de creación y depuración de la aplicación. Con el aumento progresivo de la potencia de los sistemas informáticos, la fase de compilación es cada día más rápida y permite trabajar de manera continua en esta modalidad. En los últimos tiempos, con la evolución de las redes de comunicaciones y la creación

de aplicaciones pensadas para este entorno, se está volviendo a la ejecución interpretada. Los procedimientos se envían en código de alto nivel hasta las máquinas remotas que los ejecutan mediante el uso de intérpretes. Esta modalidad tiene dos grandes ventajas: el código de alto nivel es más compacto que el código máquina y, además, es independiente de la plataforma en la que se ejecuta.

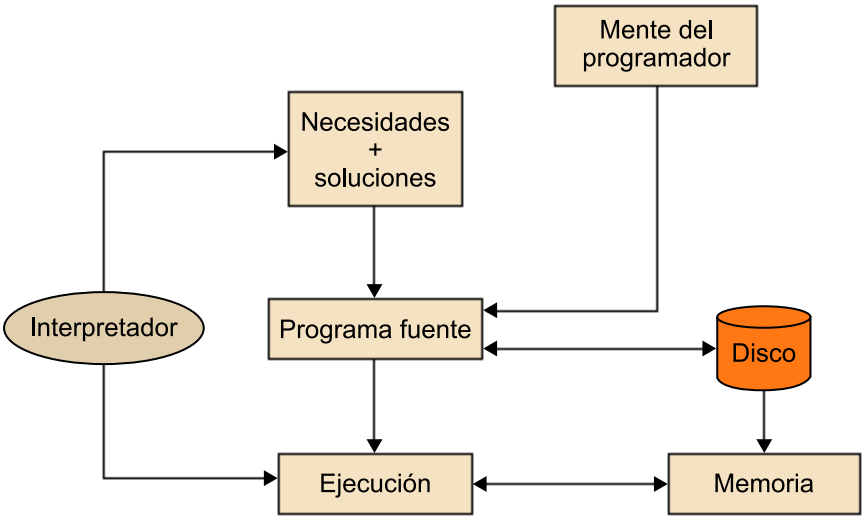

Figura 14. Proceso de interpretación

B) Los espacios de direcciones de un proceso

Ahora analizaremos el mecanismo de carga de un programa en la memoria y la manera como esta memoria se diferencia de la memoria asignada a otros procesos o de la asignada al sistema operativo mismo.

BA) El espacio lógico y el espacio físico

Partimos del espacio definido en el fichero ejecutable, que puede tener una estructura de una o dos dimensiones (espacio lineal o segmentado). Estos espacios se tienen que transportar desde los ficheros ejecutables hasta la memoria que gestiona el sistema operativo. La estructura que tiene esta memoria puede ser muy diferente en función del modelo escogido por el sistema operativo.

Un programa reubicable se puede ejecutar en direcciones diferentes, que son asignadas por el cargador. Por este motivo, se distingue entre direcciones lógicas y direcciones físicas:

- **Direcciones lógicas**: son las referenciadas por el programador. Suelen ser identificadores que reconocen las diferentes partes de un programa dentro de su espacio de direcciones. Son referencias locales en el sentido de que el entorno en el que se tengan que ejecutar no tiene importancia.

- **Direcciones físicas**: son las direcciones asignadas en tiempo de carga, es decir, el lugar donde residirá realmente el proceso durante su ejecución.

BB) La carga en la memoria: reubicación

La propiedad de la reubicación hace referencia a la capacidad de cargar y después ejecutar un programa en cualquier posición de memoria. Es la propiedad contraria a la obligatoriedad de conocer todas las direcciones de manera fija en tiempo de compilación. Obviamente, existe un mecanismo de traducción entre las direcciones lógicas y las físicas. En función de cómo se haga, podemos distinguir dos tipos de reubicación:

- **Reubicación estática**: se da antes o durante la carga del programa en la memoria para ser ejecutado. Durante el proceso de creación de un programa ejecutable, normalmente se toma la dirección lógica 0 como dirección inicial del programa. De esta manera, las demás direcciones del programa serán direcciones lógicas relativas a la dirección final de carga del programa. Cuando se carga la imagen del programa, todas las direcciones marcadas como reubicables se transforman en la dirección física final. La referencia para poder hacerlo es la dirección lógica inicial.
  Una vez se han modificado todas estas direcciones para reubicarlas, no se pueden distinguir las direcciones reubicadas de las originales, en otras palabras, no se puede volver a reubicar durante la ejecución. Si por alguna razón el proceso tiene que salir de su espacio físico asignado, o bien vuelve exactamente al mismo lugar o bien se tiene que iniciar el proceso de reubicación desde el principio, es decir, se tiene que volver a cargar.
  A causa del problema anterior, la reubicación estática está prácticamente limitada a ciertos sistemas operativos en los que el mecanismo de carga de programas en la memoria está muy determinado y el código tiene que ir a particiones de memoria predeterminadas.

- **Reubicación dinámica**: en este caso, la transformación entre las direcciones físicas y lógicas se efectúa en tiempo de ejecución. Como antes, los programas ejecutables resultantes de las compilaciones adoptan la dirección inicial de referencia 0.
  Estas imágenes de los programas ejecutables se cargan directamente en cualquier posición de memoria y es una parte del hardware el que se encarga de hacer el trabajo de reubicación de direcciones lógicas a físicas.
  Cuando el proceso se está ejecutando, antes de acceder realmente a la memoria física se reubican todos los accesos a la memoria. Este proceso se lleva a cabo utilizando los registros de base. Cada referencia que hace un proceso durante su ejecución es transformada por la suma de la dirección lógica más el registro de base para obtener la dirección física. Este mecanismo es totalmente transparente para el programador.
  La transformación de direcciones depende en gran medida del hardware del sistema. Por este motivo no podemos dar más detalles sin hablar de una

arquitectura de ordenador determinada; pero, de hecho, eso no es objeto de esta asignatura.

C) Mapa conceptual

El mapa conceptual de la figura 15 refleja las relaciones que existen entre los diferentes conceptos que hemos presentado en este módulo.

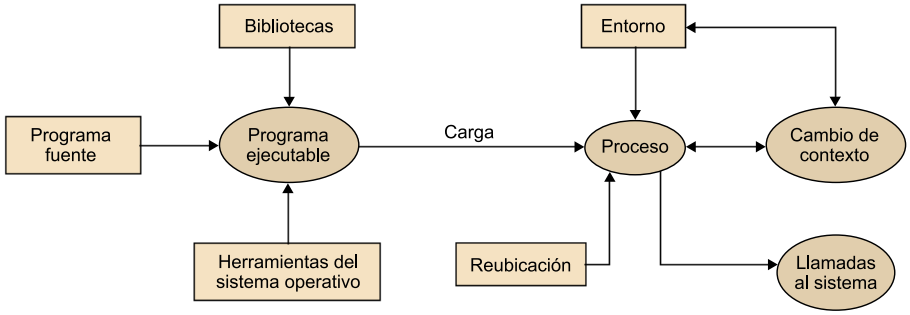

Figura 15. Mapa conceptual