

Conceptos Generales

Tabla de Contenidos

1. Conceptos Generales.....	2
1.1 Introducción:.....	2
1.2 Definición de un Sistema Operativo.....	2
1.3 Estructura, elementos y funciones.....	5
1.3.1 Evolución de los sistemas operativos:.....	5
Primera etapa: Procesamiento en serie.....	5
Segunda etapa: Procesamiento por lotes.....	6
Tercera etapa: Multiprogramación y tiempo compartido.....	7
Cuarta etapa: redes de ordenadores.....	8
1.3.2 Interfaces con el sistema operativo.....	8
Las llamadas al sistema.....	8
El intérprete de órdenes.....	9
Llamadas al sistema y protección.....	10
1.3.3 Estructura de los sistemas operativos.....	11
Sistemas monolíticos.....	11
Modelo cliente-servidor.....	13
Máquina Virtual.....	14
Estructuras orientadas a objetos.....	15
1.3.4 Clasificación de los sistemas operativos:.....	16
1.3.5 Procesos.....	17
1.3.6 Gestión de memoria.....	30
1.3.7 Gestión de discos.....	50
1.3.8 Entrada / salida.....	59
1.3.9 Compiladores.....	71



1. Conceptos Generales

1.1 Introducción:

Un ordenador es una máquina que procesa la información de forma rápida y automática. A pesar de ello, la utilización de un ordenador no es una tarea fácil, ya que la forma en que podemos comunicarnos con él, es decir, su interfaz, es extraña y compleja. Se entiende por interfaz de un objeto la parte del objeto accesible desde su exterior, que nos permite utilizarlo y consultar su estado interno. Por ejemplo, un reloj digital, cuya interfaz esta constituida por una serie de botones y una pantalla, que hacen posible la modificación su comportamiento, cambiando la hora o programando una alarma, y la visualización de su estado actual. Para usar un reloj digital no es preciso conocer su funcionamiento interno, sólo hay que saber utilizar su interfaz. La interfaz de un ordenador está formada por un conjunto pequeño de instrucciones máquina, que permiten utilizar los dispositivos físicos o hardware (CPU, memoria y periféricos) de los que se compone. Si los usuarios de un ordenador tuvieran que utilizar el hardware a través de sus instrucciones máquina se escribirían muy pocos programas, porque trabajar de manera directa con los dispositivos físicos del ordenador mediante estas instrucciones es complejo, tedioso, y está lleno de detalles.

La solución que se adoptó con el tiempo para salvar esta complejidad es la de escribir capas o niveles de software. Una capa de software de nivel i es un conjunto de programas que trabajan directamente con la interfaz de su nivel inferior ($i - 1$), y presentan a su nivel superior ($i + 1$) una interfaz que permite utilizar el nivel $i - 1$ de una forma más sencilla. Una capa software abstrae a su nivel superior de los detalles del nivel inferior y, por tanto, una capa de nivel i puede permitir a su capa superior utilizar las interfaces de sus niveles inferiores ($i - 1$, $i - 2$, ...), o puede obligar a utilizar solamente la interfaz de la capa i , asegurándose así la utilización directa del nivel inferior. El sistema operativo es una de las capas de software más importantes de un sistema informático.

1.2 Definición De Un Sistema Operativo

Se puede definir un **sistema operativo** como un conjunto de programas que controlan directamente los recursos *hardware* o físicos de un ordenador (CPU, memoria principal y periféricos) proporcionando una máquina virtual más fácil de utilizar que el *hardware* subyacente. El sistema operativo es la capa de *software* más baja de un ordenador y que trabaja directamente sobre el *hardware*.

Por encima del sistema operativo se encuentra un nivel formado por traductores, editores de texto e intérpretes de órdenes. Los dos primeros tipos de programas, junto con enlazadores y depuradores, son útiles para crear un nivel de abstracción cómodo para el desarrollo de programas.



Figura 1. Niveles hardware y software de un ordenador.

La unión de los programas de las dos capas intermedias de la Figura anterior forman el **software de sistemas** de un ordenador. Finalmente, está el nivel constituido por los programas de aplicación, estos programas no dan servicio a otros programas, su finalidad es resolver problemas concretos. Pertenecen a esta capa los procesadores de texto, hojas de cálculo, etc.

En general, puede decirse que los sistemas operativos realizan dos funciones:

1. Constitución de una máquina virtual o extendida:

El sistema operativo proporciona al usuario una **máquina virtual** cuyas características son distintas y más sencillas de utilizar, que las de la máquina real subyacente. Algunas características en las que frecuentemente la máquina virtual difiere de la máquina real son:

a) Entrada/salida (E/S).

La capacidad de E/S de un *hardware* básico es extremadamente complejo requiere sofisticados programas para su utilización. Un sistema operativo evita al usuario tener que comprender el funcionamiento de este *hardware*, poniendo a su alcance una máquina virtual mucho más sencilla.

b) Memoria.

Muchos sistemas operativos proporcionan una máquina virtual cuya memoria difiere en tamaño de la de la máquina real subyacente. Por ejemplo, un sistema operativo puede emplear memoria secundaria (discos magnéticos) para obtener una memoria principal mucho más extensa, la cual es virtual y difiere de la que se dispone en realidad.

c) Sistema de ficheros.

La mayoría de las máquinas virtuales incluyen un sistema de ficheros para el almacenamiento a largo plazo tanto de programas como de datos. El sistema de ficheros está basado en la capacidad de almacenamiento sobre cinta o disco de la máquina real. El sistema operativo, sin embargo, permite al usuario acceder a la información almacenada a través de nombres simbólicos en lugar de hacerlo a través de su posición física en el medio de almacenamiento.

d) Protección y tratamiento de errores.

En ordenadores compartidos por un determinado número de usuarios, es esencial que cada uno de ellos esté protegido de los efectos de los errores. Los ordenadores varían en lo que respecta al grado de protección que proporciona su *hardware* básico, siendo misión del sistema operativo el constituir una máquina virtual en la que ningún usuario pueda afectar de manera negativa al trabajo de los demás.

e) Interacción a nivel de programa.

Una máquina virtual puede posibilitar la interacción entre distintos programas de los usuarios de forma que, por ejemplo, la salida de uno de ellos sea la entrada de otro. La naturaleza concreta de una máquina virtual dependerá de la aplicación particular a la que se dedique. Por ejemplo, las características de una máquina virtual que controle un sistema de tiempo real serán distintas de las de una máquina virtual que se utilice para el desarrollo de programas.

2. Utilización compartida de recursos:

Un sistema operativo debe lograr que se compartan los recursos de un ordenador entre un cierto número de usuarios que trabajen de forma simultánea. La finalidad de esto es la de incrementar la disponibilidad del ordenador con respecto a sus usuarios y, al mismo tiempo, maximizar la utilización de recursos tales como procesadores centrales, memoria y dispositivos de E/S.



1.3 Estructura, Elementos Y Funciones.

1.3.1 Evolución De Los Sistemas Operativos:

Primera Etapa: Procesamiento En Serie

Al principio no existían sistemas operativos, programándose sobre el *hardware* básico. Los programas se escribían en lenguaje máquina, y se introducían en el ordenador, junto a los datos, en octal o hexadecimal mediante una consola con interruptores manuales. Se iniciaban los programas cargando el registro contador de programa con la dirección de memoria de la primera instrucción del programa. Los resultados de la ejecución se obtenían examinando el contenido de los registros y posiciones de memoria relevantes. Los dispositivos de E/S se controlaban directamente, escribiendo y leyendo en los puertos de E/S.

La programación del *hardware* básico da una baja productividad. El proceso largo y tedioso de la introducción de programas y datos excluye prácticamente la ejecución de programas medios y grandes. Posteriormente, la evolución viene con la llegada de los dispositivos de E/S, tales como lectores de tarjetas y de cintas de papel perforadas. También aparecen los traductores de lenguajes. Los programas, codificados ahora en un lenguaje simbólico, se traducen a forma ejecutable mediante un traductor. Otro programa, llamado **cargador**, automatiza la carga de programas ejecutables en memoria. El usuario pone un programa y sus datos de entrada en un dispositivo de entrada, y el cargador transfiere información desde el dispositivo de entrada a memoria. Después, se transfiere el control, mediante medios manuales o automáticos, al programa cargado para su ejecución. El programa en ejecución lee sus entradas desde el dispositivo de entrada designado y puede producir alguna salida en un dispositivo de salida, como la impresora o la pantalla. Una vez en memoria, el programa se puede reejecutar con un conjunto diferente de datos de entrada.

El mecanismo de desarrollo de programas era una secuencia típica; Se carga el programa editor para preparar el código fuente del programa de usuario. El siguiente paso es cargar y ejecutar el traductor, y alimentarlo con el código fuente del programa de usuario. Los traductores de paso múltiple pueden requerir que se vuelva a poner el código fuente durante cada paso para leerlo. Una vez corregido sintácticamente el programa se ejecuta el código objeto. Si se detectan errores en la ejecución, se puede examinar y modificar el contenido de la máquina mediante los interruptores de la consola, o con la ayuda de un programa denominado **depurador**. La mayoría de los programas utilizaban dispositivos de E/S. Una mejora lógica fue el proporcionar unas rutinas estándares de E/S que fueran usadas

por todos los programas. Al principio, las rutinas de E/S se introducían con las demás tarjetas del programa de usuario. Posteriormente, se guardaba en memoria las rutinas compiladas, y mediante un programa **enlazador** se combinaban con el código objeto del usuario. En estos sistemas las rutinas de E/S y el programa cargador constituyen una forma rudimentaria de sistema operativo. Los traductores, editores y depuradores son programas de sistema, pero no forman parte del sistema operativo.

Segunda Etapa: Procesamiento Por Lotes

Anteriormente, el rendimiento del procesador es muy baja, porque el tiempo empleado en leer un programa almacenado en tarjetas suele ser mucho mayor que el empleado en ejecutar el programa. Cuando aparecieron las cintas magnéticas, cuya lectura y escritura era muy inferior en tiempo a las tarjetas, se pensó que se utilizaría más el procesador si todas las entradas y salidas se realizaban sobre cintas. Para realizar esto se utilizó una técnica de **off-lining (fuera de línea)**. Se dedicaba un ordenador periférico, de menor costo y potencia, a convertir las tarjetas o la cinta perforada en información sobre cinta magnética, y la salida sobre cinta magnética en salida sobre impresora o cinta perforada. Una vez que se procesaban varios trabajos a cinta, ésta se desmontaba del ordenador periférico, y se llevaba a mano para su procesamiento por el ordenador principal. Cuando el ordenador principal llenaba una cinta de salida, ésta se llevaba al ordenador periférico para su paso a impresora o cinta perforada. Una de las implicaciones de esta forma de trabajo era que en una cinta de entrada podían existir los trabajos de varios programadores. Con esta forma de trabajo se esperaban horas hasta que el programa proporcionara su salida. La falta de un punto y coma al final de una línea de un programa podía provocar un error sintáctico, y la pérdida de estas horas de espera. Por otro lado, debido a que la cinta magnética es un medio de almacenamiento serie, no había opción alguna de orden de ejecución de las tareas que no fuese el orden en que éstas se habían presentado al ordenador.

De cara a eliminar la dependencia de las E/S en lugar de tan sólo reducirla, hay que emplear técnicas mediante las cuales se puedan superponer las E/S al proceso a ejecutar. Ello es posible con la ayuda de dos elementos del **hardware**: el **canal** y la **interrupción**. Un canal es un elemento que controla uno o más dispositivos, llevando a cabo transferencias de datos entre estos dispositivos y la memoria sin involucrar prácticamente al procesador central. Una interrupción es una señal que transfiere el control del procesador central a una posición fija de memoria, almacenando al mismo tiempo el valor anterior del contador de programa, y, posiblemente, la palabra de estado del procesador. De esta forma, se suspende temporalmente la ejecución del programa que estaba siendo llevado a cabo en el momento de la interrupción, aunque podrá reemprenderse dicha ejecución más tarde (o sea, el programa es **interrumpido**). Una interrupción de un canal actúa como una señal que indica que se ha completado una transferencia de datos. De esta forma es posible que el procesador central inicie una transferencia a un dispositivo, continúe el proceso que estaba llevando a cabo mientras el canal controla la transmisión y reciba a través de una

interrupción la notificación de haberse completado dicha transferencia. Es posible ahora leer las tareas a ejecutar guardándolas en un soporte adecuado (normalmente disco), y ejecutarlas una a una al mismo tiempo que se van leyendo otras. Para ello ha habido que añadir a nuestro sistema operativo una rutina de gestión de las interrupciones y otra que decida cuál de las tareas almacenadas en disco será la siguiente en ser ejecutada. Esta última función, que recibe el nombre de *scheduling*, deriva del empleo del disco (caracterizado por un acceso aleatorio) como medio de almacenamiento de las distintas tareas en lugar de la cinta magnética, caracterizada por un acceso serie. Un sistema que trabaje de esta forma recibe el nombre de **monitor de batch de flujo único** (en inglés, *single stream batch monitor*). El concepto de flujo único lleva implícita la idea de una sola tarea ejecutándose a la vez.

Tercera Etapa: Multiprogramación Y Tiempo Compartido

La desventaja de un sistema de flujo único es la total dedicación de la máquina a la ejecución de una sola tarea, no importa lo larga o lo corta que sea. Este inconveniente se soluciona mediante la **multiprogramación**, que es la ejecución simultánea de varios programas que residen en la memoria principal, dividiendo el procesador central su tiempo entre ellos de acuerdo con los recursos (canales o dispositivos) que necesite en cada momento cada uno de ellos. De esta forma es posible, teniendo almacenado un conjunto adecuado de tareas en cada momento, obtener una utilización óptima de los recursos disponibles. Ello incluye la utilización del procesador central, ya que en tanto que una tarea esté esperando el final de una transferencia de E/S, este procesador puede pasar a trabajar en alguna otra tarea que esté pendiente en la máquina. La carga que recae sobre el sistema operativo consiste en el control de los recursos, así como la protección de cada tarea frente a las actividades de las otras. Un sistema operativo de este tipo recibe el nombre de **monitor de batch de varios flujos**. Sin embargo, desde el punto de vista del usuario el sistema posee falta de interactividad. Para hacer posible esta interacción, el sistema de *batch* de varios flujos debe modificarse con el fin de que pueda adquirir la información que le suministren los usuarios desde los respectivos terminales: es decir, debe convertirse en un sistema **multiusuario**. Un sistema de este tipo, en el cual existen varios usuarios con un terminal en línea (usuarios interactivos), se llama sistema de **tiempo compartido**. En estos sistemas se divide el tiempo del procesador central, y de los demás recursos del ordenador, de forma que cada usuario tiene la ilusión de que todo el ordenador se le dedica exclusivamente a él, al recibir unos tiempos de respuesta aceptables. En un sistema de tiempo compartido los usuarios suelen ejecutar programas cortos (por ejemplo, compilar un programa), frente a los programas largos (por ejemplo, ordenar una cinta de un millón de datos) que se suelen dar en los sistemas *batch*. Por último hay que indicar que algunos sistemas operativos permiten tanto usuarios interactivos como lotes de trabajos *batch*. En estos sistemas se atiende a los usuarios interactivos con mayor prioridad, ejecutándose los programas *batch* cuando no hay programas de usuario.



Cuarta Etapa: Redes De Ordenadores

En una red de ordenadores se tiene una configuración de varios ordenadores conectados físicamente. Los ordenadores de una red pueden tener **sistemas operativos de red** o **sistemas operativos distribuidos**. En un sistema operativo de red los usuarios son conscientes de la existencia de varios ordenadores, y pueden conectarse con máquinas remotas para, por ejemplo, copiar ficheros. Cada máquina ejecuta su propio sistema operativo local y tiene su propio usuario (o grupo de usuarios). Los sistemas operativos de red no difieren de los sistemas operativos tradicionales de un sólo procesador. Necesitan un controlador de red, algunas rutinas de E/S para utilizar dicho controlador, y programas que permitan la conexión y el acceso a ordenadores remotos, pero esas características adicionales no modifican la estructura esencial del sistema operativo. En un sistema distribuido los ficheros que utiliza un usuario (así como el procesador en el que se ejecutan sus programas) pueden estar situados en cualquier ordenador de la red. Además, esto es transparente al usuario. Los sistemas operativos distribuidos requieren algo más que añadir un poco de código a un sistema operativo de un único procesador, ya que los sistemas distribuidos difieren en aspectos críticos de los sistemas centralizados. Actualmente toman fuerza nuevas tendencias como el *Multiprocesamiento Simétrico*, donde todas las CPU's que forman el sistema comparten una única copia del código y los datos del kernel, frente a *Multiprocesamiento Asimétrico*, donde cada CPU tiene su propio kernel y datos locales.

1.3.2 Interfaces Con El Sistema Operativo

Las Llamadas Al Sistema

Anteriormente se comentó que el sistema operativo es una interfaz que oculta las peculiaridades del *hardware*. Para ello ofrece una serie de servicios que constituyen una máquina virtual más fácil de usar que el *hardware* básico. Estos servicios se solicitan mediante **llamadas al sistema**, consistente en colocar una serie de parámetros en un lugar específico (como los registros del procesador), para después ejecutar una instrucción del lenguaje máquina del procesador denominada *trap*, en castellano, **trampa**. La ejecución de esta instrucción máquina hace que el hardware guarde el contador de programa y la palabra de estado del procesador en un lugar seguro de la memoria, cargándose un nuevo contador de programa y una nueva palabra de estado del procesador. Este nuevo contador de programa contiene una dirección de memoria donde reside una parte (un programa) del sistema operativo, el cual se encarga de llevar a cabo el servicio solicitado. Cuando el sistema operativo finaliza el servicio, coloca un código de estado en un

registro para indicar si hubo éxito o fracaso, y ejecuta una instrucción ***return from trap***, esta instrucción provoca que el *hardware* restituya el contador de programa y la palabra de estado del procesador del programa que realizó la llamada al sistema, prosiguiéndose así su ejecución. Normalmente los lenguajes de alto nivel tienen una (o varias) rutinas de biblioteca por cada llamada al sistema. Dentro de estos procedimientos se aísla el código (normalmente en ensamblador) correspondiente a la carga de registros con parámetros, a la instrucción *trap*, y a obtener el código de estado a partir de un registro. La finalidad de estos procedimientos de biblioteca es ocultar los detalles de la llamada al sistema, ofreciendo una interfaz de llamada a procedimiento. Como una llamada al sistema depende del *hardware* (por ejemplo, del tipo de registros del procesador), la utilización de rutinas de biblioteca hace el código portable. El número y tipo de llamadas al sistema varía de un sistema operativo a otro. Existen, por lo general, llamadas al sistema para ejecutar ficheros que contienen programas, pedir más memoria dinámica para un programa, realizar labores de E/S, etc.

El Intérprete De Órdenes

Cuando un usuario se conecta a un ordenador se inicia un intérprete de órdenes, en entornos UNIX, son llamados ***shells***. El intérprete de órdenes es un programa que muestra un **indicador (*prompt*)** formado por algunos caracteres, que pueden incluir el directorio de trabajo, que indica al usuario que es posible introducir una orden. En un entorno UNIX, o MS DOS, la primera palabra es el nombre de un fichero que contiene un programa, siendo el resto de la línea una serie de argumentos, separados por espacios, que toma dicho programa. Una excepción a esto son las órdenes internas, que el intérprete implementa como rutinas suyas, y que no tienen, por tanto, un programa asociado guardado en disco. El intérprete de órdenes realiza entonces una o varias llamadas al sistema para ejecutar dicho programa. Cuando el programa finalice (al realizar una llamada al sistema *exit*) el control vuelve al programa que lo lanzó (el intérprete de órdenes), mostrando éste otra vez el indicador, y repitiéndose el ciclo. Así pues, el intérprete de órdenes es un programa que sirve de interfaz entre el sistema operativo y un usuario, utilizándolo este último para ejecutar programas.

A diferencia de un programador, un usuario final realiza todas las llamadas al sistema indirectamente, a través de las llamadas al sistema de los programas que ejecuta. En muchos sistemas se ha optado por sustituir el intérprete de órdenes por un programa que utiliza ventanas, como en el caso de sistemas operativos Windows. En estos programas aparecen iconos que el usuario puede seleccionar mediante un ratón. Cuando el usuario selecciona un icono que representa un programa se realizan llamadas al sistema para ejecutar un fichero, asociado al icono, que contiene el programa. Por lo tanto, se sustituye la interfaz del usuario para ejecutar programas, pero la interfaz con el sistema operativo no cambia. Existen una serie de programas muy importantes (como traductores, editores de texto, ensambladores, enlazadores e intérpretes de órdenes) que ayudan al programador a realizar sus programas, y que

vienen en el lote con cualquier sistema operativo. Estos programas, que forman parte de los **programas de sistema** o **software de sistemas**, utilizan llamadas al sistema, pero no son parte del sistema operativo. El sistema operativo es el código que acepta llamadas al sistema, y realiza un procesamiento para satisfacer dichas llamadas.

Llamadas Al Sistema Y Protección

En los sistemas de multiprogramación se pueden ejecutar varios programas a la vez. Cuando se ejecutan simultáneamente varios programas, por ejemplo de distintos usuarios, hay que proteger a un programa de la acción de los demás. Para llevar a cabo esta protección, el sistema operativo se apoya en varios mecanismos proporcionados por el *hardware*. Los mecanismos de protección de memoria se verán más adelante en la de gestión de memoria.

La mayoría de los procesadores tienen un **modo supervisor** (o **modo núcleo**) y el **modo usuario**, un bit de la palabra de estado del procesador indica el modo de funcionamiento. En modo supervisor está permitido la ejecución de cualquier instrucción máquina, sin embargo, en modo usuario no se permite la ejecución de algunas instrucciones reservadas que llevan a cabo funciones tales como:

1. autorizar e inhibir las interrupciones.
2. conmutar un procesador entre distintos procesos (programa en ejecución).
3. acceder a los registros utilizados por el *hardware* de protección de la memoria.
4. realizar operaciones de E/S.
5. parar la CPU.

El sistema operativo se ejecuta en modo supervisor con acceso total al *hardware*, mientras que los programas de usuario se ejecutan en modo usuario. Cuando un programa necesita utilizar un recurso *hardware*, por ejemplo, quiere realizar una operación de E/S, debe realizar una llamada al sistema para que el sistema operativo lleve a cabo la operación por él. La realización de una llamada al sistema conmuta automáticamente la CPU a modo supervisor. Así se garantiza que, por ejemplo, un usuario al que no se le ha concedido una impresora utilice instrucciones máquina para acceder directamente al controlador de la impresora, interfiriendo con el programa al que se le concedió la impresora. La vuelta al modo usuario desde el modo supervisor se lleva a cabo mediante una instrucción, también reservada.

1.3.3 Estructura De Los Sistemas Operativos

En este apartado se analizará el interior del sistema operativo, examinando las formas posibles de estructurar el código de un sistema operativo.

Sistemas Monolíticos

Este tipo de organización es la más común. El sistema operativo se escribe como una colección de procedimientos, cada uno de los cuales puede llamar a los demás cada vez que así lo requiera. Cuando se usa esta técnica, cada procedimiento del sistema tiene una interfaz bien definida en términos de parámetros y resultados, y cada uno de ellos es libre de llamar a cualquier otro, si éste último proporciona un cálculo útil para el primero. Para construir el programa objeto real del sistema operativo siguiendo este punto de vista, se compilan de forma individual los procedimientos, o los ficheros que contienen los procedimientos, y después se enlazan en un sólo fichero objeto con el enlazador. La ocultación de la información es prácticamente nula, puesto que cada procedimiento es visible a los demás.

Las llamadas al sistema que proporciona el sistema operativo se solicitan colocando los parámetros en lugares bien definidos, como los registros o la pila, para después ejecutar una instrucción especial de trampa, a veces referida como **llamada al núcleo** o **llamada al supervisor**. Esta instrucción cambia la máquina del modo usuario al modo supervisor y transfiere el control al sistema operativo (evento 1 de la Figura 2). El sistema operativo examina entonces los parámetros de la llamada para determinar cual de ellas se desea realizar (evento 2 de la Figura 2). A continuación, el sistema operativo analiza una tabla que contiene en la entrada k un puntero al procedimiento que implementa la k -ésima llamada al sistema (evento 3 de la Figura 2), identifica el procedimiento de servicio, al cual se llama. Por último, la llamada al sistema termina y el control vuelve al programa del usuario.



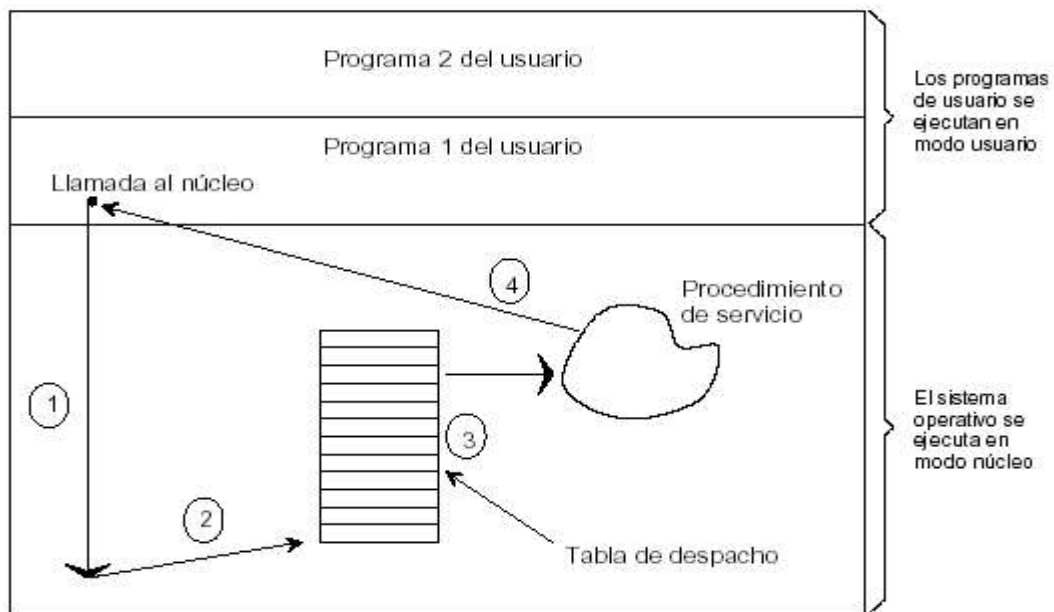


Figura 2. Llamada al sistema.

Esta organización sugiere una estructura básica del sistema operativo:

1. Un programa principal que llama al procedimiento del servicio solicitado.
2. Un conjunto de procedimientos de servicio que lleva a cabo las llamadas al sistema.
3. Un conjunto de procedimientos de utilidades que ayudan a los procedimientos de servicio.

Modelo Cliente-servidor

Los sistemas operativos modernos trasladan el código a capas superiores y eliminan la mayor parte posible del sistema operativo para mantener un núcleo mínimo. Generalmente, se implantan la mayoría de las funciones del sistema operativo como procesos de usuario. Para solicitar un servicio, como la lectura de un bloque de cierto fichero, un proceso de usuario, ó **proceso cliente**, envía la solicitud a un **proceso servidor**, que realiza el trabajo y devuelve la respuesta. En este modelo lo único que hace el núcleo es controlar la comunicación entre los clientes y los servidores. Al separar el sistema operativo en partes, cada una de ellas controla un aspecto del sistema, como el servicio a ficheros, servicio a procesos, servicio a terminales o servicio a la memoria; cada aspecto es pequeño y controlable. Además, puesto que todos los servidores se ejecutan como procesos en modo usuario, y no en modo núcleo, no tienen acceso directo al *hardware*. En consecuencia, si hay un error en el servidor de ficheros éste puede fallar, pero esto no afectará en general a toda la máquina.

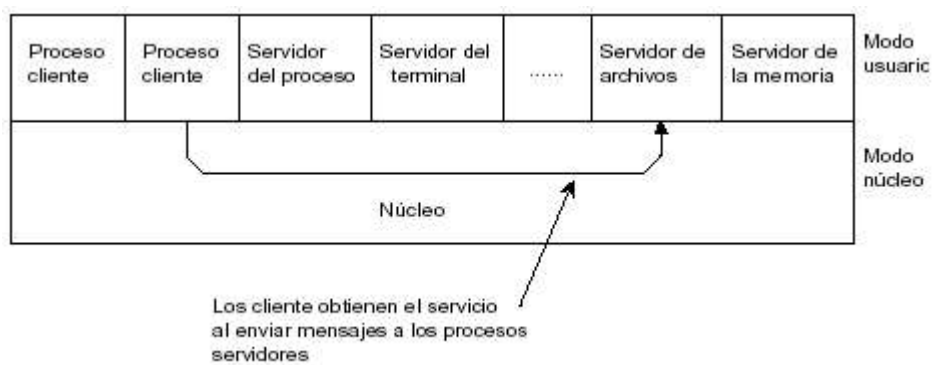


Figura 3. El modelo cliente-servidor

Otra ventaja del modelo cliente-servidor es su capacidad de adaptación para su uso en sistemas distribuidos. Si un cliente se comunica con un servidor mediante mensajes, el cliente no necesita saber si el mensaje se gestiona de forma local, en su máquina, o si se envía por medio de una red a un servidor en una máquina remota. La idea anterior de un núcleo que sólo controla el transporte de mensajes de clientes a servidores, y viceversa, no es totalmente real. Algunas funciones del sistema operativo, como la introducción de órdenes en los registros físicos de los controladores de E/S, son difíciles o imposibles de realizar, a partir de programas de usuario. La solución radicaría en hacer que algunos procesos de servidores críticos (por ejemplo, los gestores de los dispositivos de E/S) se ejecuten en realidad en modo núcleo, con acceso total al *hardware*, pero de forma que se comuniquen con los demás procesos mediante el mecanismo normal de mensajes.

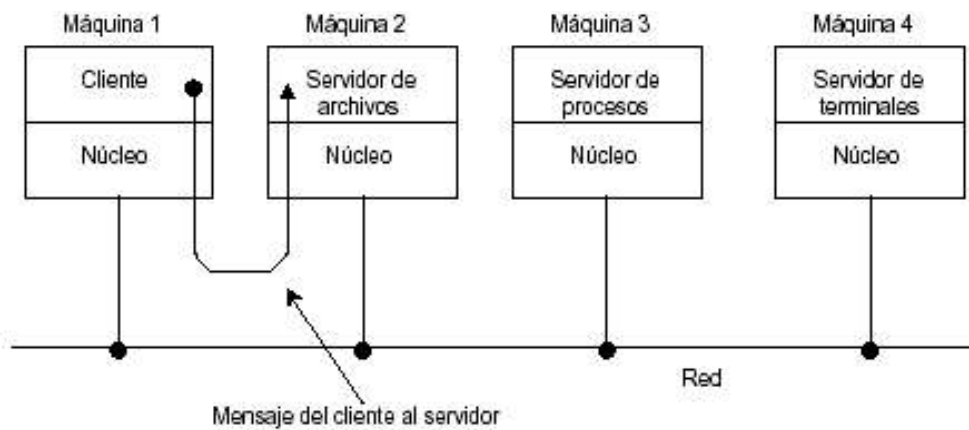


Figura 4. El modelo cliente-servidor en un sistema distribuido.

Máquina Virtual

Esta estructura proporciona un conjunto de máquinas abstractas, cada una de las cuales actúa casi como una copia idéntica del hardware subyacente. Esta estructura funciona simulando individualmente cada máquina abstracta sobre la máquina base real. Así, para cada usuario del sistema, este aparece como si el tuviera su propia copia del hardware subyacente, de forma que obtiene la protección y seguridad de manera directa. El problema de esta estructura es que el rendimiento del sistema operativo se va degradando, puesto que la simulación es costosa, mientras que la principal ventaja es que permite implementar varios sistemas operativos sobre cada máquina virtual. Sin embargo, los sistemas operativos de cada máquina virtual son disjuntos, por lo que no se podrían utilizar las interacciones, compartimiento, y comunicación necesarios en los sistemas actuales.

Estructuras Orientadas A Objetos

En ellas los servicios del sistema operativo se implementan como una colección de objetos, los cuales se definen como segmentos protegidos por capacidades. Cada objeto tiene un tipo que designa sus propiedades: procesos, directorios, archivos, etc. Un objeto tiene un conjunto de operaciones mediante las cuales se puede acceder y modificar su segmento interno. Antes de que un usuario solicite un servicio de un objeto, este debe adquirir sus capacidades, incluidos los derechos de las operaciones permitidas. El kernel tiene la responsabilidad de proteger las capacidades frente a accesos erróneos o malintencionados.



1.3.4 Clasificación De Los Sistemas Operativos:

Los sistemas operativos se pueden clasificar atendiendo a diferentes criterios, como son:

- **Modo de trabajo del usuario:**

Se pueden clasificar en *on line* (o interactivos) y *off line* (o *batch* o por lotes). Un ejemplo del primer tipo son los sistemas de tiempo compartido, los sistemas interactivos son útiles, entre otros, en entornos de desarrollo de programas, de procesamiento de textos y de ejecución de programas interactivos. Un ejemplo del segundo tipo son los sistemas por lotes. Los sistemas *batch* se caracterizan porque una vez introducida una tarea en el ordenador, el usuario no mantiene contacto alguno con ella hasta que finaliza su ejecución.

- **Número de usuarios:**

Atendiendo al número de usuarios se encuentran los sistemas **monousuario** y **multiusuario**. En el primer tipo se puede acceder al ordenador mediante un único terminal, mientras que los segundos permiten varios terminales de acceso simultáneo. Un ejemplo muy claro de sistemas operativos monousuario es MS DOS y CP/M. Ejemplos de sistemas multiusuario son UNIX y AS/400.

- **Propósito del usuario:**

En base al uso que quieran dar los usuarios al ordenador, se encuentran sistemas operativos de **propósito específico** y de **propósito general**. Un ejemplo de sistema de propósito específico es un sistema de tiempo real, estos sistemas se usan en entornos donde se deben aceptar y procesar en tiempo breve un gran número de sucesos como por ejemplo aplicaciones para el equipamiento telefónico conmutado o para control de vuelo. Los sistemas de propósito general se caracterizan por el gran número de usuarios trabajando sobre un amplio abanico de aplicaciones.

- **Número de procesadores:**

Los ordenadores con varias CPU's se clasifican en **multiprocesadores** y en **sistemas distribuidos**. En un multiprocesador los procesadores comparten memoria y reloj, puesto que son asíncronos, mientras que en un sistema distribuido tenemos varios procesadores con su propia memoria y además no están sincronizados.



1.3.5 Procesos.

Los sistemas operativos multiprogramados necesitan el concepto de **proceso**. El sistema operativo debe entremezclar la ejecución de un número de procesos para aumentar el rendimiento de los recursos del ordenador, a la vez que los sistemas de tiempo compartido deben proporcionar un tiempo de respuesta razonable, por lo que el sistema operativo debe asignar recursos a los procesos de acuerdo con una política específica, mientras impide los interbloqueos. Además, el sistema operativo debe ofrecer un soporte para llevar a cabo la comunicación entre procesos.

Definición de proceso

Un **programa** es una secuencia de instrucciones escrita en un lenguaje, mientras que un **proceso** es una instancia de ejecución de un programa, con sus propios contador de programa, palabra de estado, registros del procesador, datos, etc. Un programa puede ser ejecutado por varios usuarios en un sistema multiusuario, por cada una de estas ejecuciones existirá un proceso, con su contador de programa, registros, etc. El sistema operativo necesita el concepto de proceso para poder gestionar el procesador mediante la técnica de multiprogramación o de tiempo compartido, de hecho, el proceso es la unidad planificable, o de asignación de la CPU.

Estados de un proceso

Un proceso puede pasar por una serie de estados discretos, algunos son:

- **En ejecución:** El proceso se está ejecutando y ocupa la CPU en ese instante.
- **Listo o preparado:** El proceso dispone de todos los recursos para su ejecución, pero tan sólo le falta la CPU.
- **Bloqueado:** Al proceso le falta algún recurso para poder seguir ejecutándose, además de la CPU. Por recurso se pueden entender un dispositivo, un dato, etc. El proceso necesita que ocurra algún evento que le permita poder proseguir su ejecución.



Transiciones de estado de los procesos

A continuación se dan ejemplos de eventos que pueden provocar transiciones de estado en un proceso en este modelo de tres estados:

- *En ejecución a Bloqueado*: al iniciar una operación de E/S, al realizar una operación WAIT sobre un semáforo a cero, al solicitar un recurso no compatible asignado a otro proceso.
- *En ejecución a Listo*: por ejemplo, en un sistema de tiempo compartido, cuando el proceso que ocupa la CPU lleva demasiado tiempo ejecutándose continuamente (agota su cuanto), el sistema operativo decide que otro proceso ocupe la CPU, pasando el proceso que ocupaba la CPU a estado listo.
- *Listo a en ejecución*: cuando lo requiere el planificador de la CPU.
- *Bloqueado a Listo*: se dispone del recurso por el que se había bloqueado el proceso. Por ejemplo, termina la operación de E/S, o se produce una operación SIGNAL sobre el semáforo en que se bloqueó el proceso, no habiendo otros procesos bloqueados en el semáforo.

Es importante ver, que de las cuatro transiciones de estado posibles, la única iniciada por el proceso de usuario es el bloqueo, las otras tres son iniciadas por entidades externas al proceso.

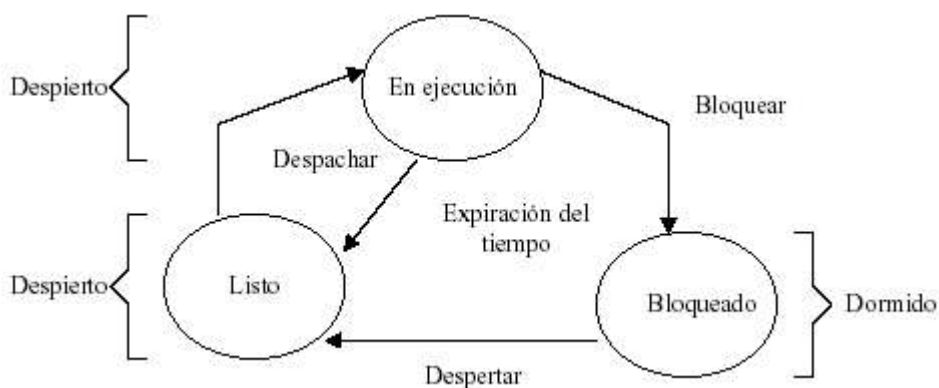


Figura 5. Transiciones de estado de los procesos.

El bloque de control de proceso (PCB)

Asociado a cada proceso hay una serie de atributos que utiliza el sistema operativo para el control del proceso. Estos atributos se agrupan en una estructura de datos que se conoce como **bloque de control de proceso** (*Process Control Block*, PCB) o **descriptor de proceso**. Al conjunto formado por el programa, datos, pila y atributos se le llama **imagen del proceso**.

El bloque de control de proceso es la estructura de datos central más importante de un sistema operativo. Cada bloque de control de proceso contiene toda la información de un proceso que necesita un sistema operativo para su control. Estos bloques son leídos y/o modificados por casi todos los módulos de un sistema operativo, especialmente para la planificación, la asignación de recursos, el tratamiento de interrupciones y el análisis y supervisión del rendimiento. En definitiva, el conjunto de los bloques de control de procesos definen el estado del sistema operativo. El conjunto de todos los PCB's se guarda en una estructura del sistema operativo llamada **tabla de procesos**, la cual se implementa como un vector o una lista enlazada. La **tabla de procesos** reside en memoria principal, debido a la alta frecuencia de su consulta.

En un sistema de multiprogramación, se necesita una gran cantidad de información de cada proceso para su administración. Esta información se organiza de modo diferente. En general, la información de los PCB's se agrupa en tres categorías:

- Identificación del proceso.
- Información de estado del procesador.
- Información de control del proceso.

La *identificación del proceso* se asigna a cada proceso un identificador numérico único (ID), para localizar al proceso dentro de la tabla de procesos. Cuando se permite que los procesos creen otros procesos, se utilizan identificadores para señalar al padre y a los descendientes de cada proceso. Además, un proceso también puede tener un identificador de usuario, que indica a quién pertenece el proceso (UID).

La *información de estado del procesador*, está formada por el contenido de los registros del procesador. Cuando se interrumpe el proceso, toda la información de los registros debe salvarse de forma que pueda restaurarse cuando el proceso reanude su ejecución. Normalmente, en el conjunto de registros se incluyen los registros visibles para el usuario, los registros de control y de estado (contador de programa y palabra de estado) y los punteros pila.

La *información de control de proceso* es la información adicional necesaria para que el sistema operativo controle y coordine los diferentes procesos activos, por ejemplo, la información de planificación y estado (estado del proceso, su prioridad, información de planificación, suceso), punteros a estructuras de datos (los procesos que esperan en un semáforo), punteros a zonas de memoria del proceso, recursos controlados por el proceso (archivos abiertos), etc.

En resumen, el PCB es la entidad que define un proceso en el sistema operativo.



Operaciones con procesos

Los sistemas que administran procesos deben ser capaces de realizar ciertas operaciones sobre y con los procesos, como:

- crear y destruir un proceso.
- suspender y reanudar un proceso.
- cambiar la prioridad de un proceso.
- bloquear y desbloquear un proceso.
- planificar un proceso (asignarle la CPU).
- permitir que un proceso se comunique con otro.

Crear un proceso implica muchas operaciones, como:

- buscarle un identificador
- insertarlo en la tabla de procesos
- determinar la prioridad inicial del proceso
- crear el PCB
- asignar los recursos iniciales al proceso

Un proceso puede crear un nuevo proceso. Si lo hace, el proceso creador se denomina **proceso padre**, y el proceso creado, **proceso hijo**. Sólo se necesita un padre para crear un hijo. Esto da lugar a la aparición de una **estructura jerárquica de procesos**, en la que cada hijo tiene sólo un padre, pero un padre puede tener muchos hijos. En el sistema operativo UNIX la llamada al sistema *fork* crea un proceso hijo.

Destruir un proceso implica eliminarlo del sistema. Se borra de las tablas o listas del sistema, sus recursos se devuelven al sistema y su PCB se borra. La destrucción de un proceso es más difícil cuando éste ha creado otros procesos. En algunos sistemas un proceso hijo se destruye automáticamente cuando su padre es destruido, mientras que en otros sistemas son independientes de su padre.

Al modelo anterior de tres estados (Listo, En ejecución y Bloqueado) se añaden un nuevo estado el *suspendido*. La suspensión de un proceso implica su desalojo de la memoria principal y su transferencia a disco. Un proceso

suspendido no puede proseguir sino hasta que lo **reanuda** otro proceso. Muchas veces, el sistema efectúa las suspensiones para eliminar temporalmente ciertos procesos, y así reducir la carga del sistema durante una situación de carga máxima. Cuando hay suspensiones largas se debe liberar los recursos del proceso. La memoria principal debe ser liberada de inmediato cuando se suspenda un proceso. Reanudar (o activar) un proceso implica reiniciarlo a partir del punto en el que se suspendió. Cambiar la prioridad de un proceso normalmente no implica más que modificar el valor de la prioridad en el PCB.

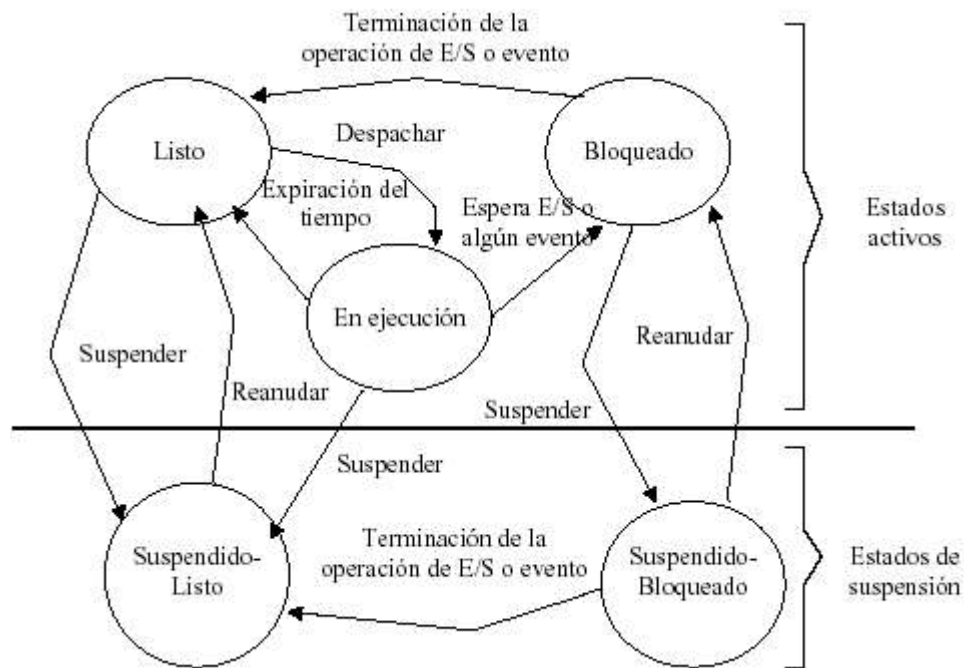


Figura 6. Transiciones de estado de los procesos.

Control de un proceso

Modos de Ejecución

Ciertas instrucciones máquina pueden ejecutarse sólo en modo privilegiado. Entre éstas están la lectura o modificación de registros de control (como la palabra de estado), instrucciones primitivas de E/S e instrucciones relativas a la gestión de memoria. Y solamente se puede acceder a ciertas zonas de memoria en el modo privilegiado. El modo de menor privilegio se conoce como modo usuario, y el de mayor privilegio como modo de sistema, supervisor o núcleo.

Se usan dos modos porque es necesario proteger al sistema operativo y a las estructuras de datos importantes, tales como los bloques de control de procesos, de las inferencias de los programas de usuario. En el modo núcleo, el *software* tiene control completo del procesador y de todas las instrucciones, registros y memoria. El procesador conoce en que modo se está ejecutando por un bit en la palabra de estado del procesador que indica el modo de ejecución. El bit se cambia como respuesta a ciertos sucesos tales como una llamada al sistema.

Cambio de Proceso

En cierto momento, un proceso que se está ejecutando se interrumpe, el sistema operativo pone a otro proceso en el estado de ejecución y pasa el control a dicho proceso. Para comprender mejor este aspecto hay que aclarar la siguiente cuestión: ¿Qué eventos provocan el cambio de proceso?. Un cambio de proceso puede suceder en cualquier instante en el que el sistema operativo gana el control de la CPU. En primer lugar, se van a tener en cuenta las interrupciones del sistema. Se pueden distinguir dos clases de interrupciones del sistema. La primera es originada por algún tipo de suceso que es externo e independiente del proceso que se está ejecutando, como la terminación de una E/S. La segunda tiene que ver con una condición de error o excepción generada dentro del proceso que se está ejecutando, como un intento ilegal de acceso a un fichero, una división entre cero, una instrucción máquina con código de operación no contemplado. En una interrupción ordinaria, el control se transfiere primero al gestor de interrupciones, quien lleva a cabo algunas tareas básicas y, después, se salta a la rutina del sistema operativo que se ocupa del tipo de interrupción que se ha producido. Algunos ejemplos de estas interrupciones son:

Interrupción de reloj: Un reloj es un dispositivo que genera interrupciones periódicamente. Ante una interrupción de este, el sistema operativo de tiempo compartido determina si el proceso en ejecución ha alcanzado el máximo tiempo de ejecución que se le concedió (cuanto). Si es así, el proceso pasará a estado listo, y se asignará la CPU a otro proceso.

Interrupción de E/S: El sistema operativo determina exactamente qué acción de E/S ha ocurrido. Si se trata de un evento por el que esperaban uno o más procesos, entonces el sistema operativo traslada todos los procesos bloqueados en dicho evento al estado listo, y determina si reanuda la ejecución del proceso interrumpido o pasa a otro de mayor prioridad.

Fallo de memoria: Un proceso hace una referencia a una dirección que no se encuentra en memoria y que debe traerse de memoria secundaria. Después de hacer la solicitud de E/S para traer esa o esas direcciones de memoria, el sistema operativo lleva a cabo un cambio de contexto para reanudar la ejecución de otro proceso; el proceso que cometió el fallo de memoria se pasa al estado bloqueado. Después de que las direcciones aludidas se carguen en memoria, dicho proceso se pondrá en estado listo. En una interrupción del segundo tipo, el sistema operativo determina si el error es fatal. Si lo es, el proceso que se estaba ejecutando es eliminado, y se produce un cambio de proceso. Si no es fatal, la acción del sistema operativo dependerá de la naturaleza del error y del diseño del sistema operativo. Se puede hacer un cambio de proceso o, simplemente, reanudar el mismo proceso que se estaba ejecutando. Finalmente, el sistema operativo puede activarse mediante una **llamada al sistema** desde el programa que se está ejecutando.

Cambio de Contexto

Si existe una interrupción pendiente es necesario:

Salvar el contexto (PC, registros del procesador, información de la pila) del programa en ejecución.

Poner la dirección del programa de tratamiento de la interrupción en el bus de direcciones, que suele constar de unas pocas tareas básicas.

Lo que constituye el contexto que se debe salvar es la información que pueda ser necesaria para reanudar el programa interrumpido. Por lo que, debe guardarse la parte del bloque de control del proceso denominada información de estado del procesador. Esto incluye al contador de programa, otros registros del procesador y la información de la pila.

La rutina de tratamiento de la interrupción es normalmente un programa corto que lleva a cabo unas pocas tareas básicas relacionadas con una interrupción. Por ejemplo, si la interrupción está relacionada con un suceso de E/S, el gestor de interrupciones comprobará condiciones de error. Si se ha producido un error, la rutina de tratamiento puede enviar una señal al proceso que solicitó originalmente la operación de E/S. La interrupción puede ir seguida de un cambio de proceso o no. La ocurrencia de una interrupción no siempre causa el cambio de proceso. Es posible que después de que el gestor de interrupciones se haya ejecutado, el proceso que estaba ejecutándose reanude su ejecución. En ese caso, tan sólo hay que guardar la información de estado del procesador y restaurarla para que pueda reanudarse correctamente el proceso interrumpido.

Por tanto, el cambio de contexto es un concepto distinto al cambio de proceso. Puede ocurrir un cambio de contexto sin cambiar el estado del proceso que está actualmente en estado de ejecución. En tal caso, salvar el contexto y restaurarlo posteriormente involucra un pequeño coste extra. Sin embargo, si el proceso que estaba ejecutándose tiene que pasar a otro estado (listo o bloqueado), el sistema operativo tiene que llevar a cabo cambios importantes en su entorno.

Los pasos de un cambio completo de proceso son los siguientes:

1. Salvar el contexto del procesador, incluyendo el contador de programa y otros registros.
2. Actualizar el PCB que estaba en estado de ejecución. Esto implica cambiar el estado del proceso a alguno de los otros estados (listo, bloqueado, suspendido_listo).
3. Mover el PCB a la cola apropiada (listos, bloqueados por el suceso *i*, *suspendido_listo*).
4. Seleccionar otro proceso para su ejecución.
5. Actualizar el PCB del proceso seleccionado.
6. Actualizar las estructuras de datos de gestión de la memoria.
7. Restaurar el contexto del procesador a aquél que existía en el momento en el que el proceso seleccionado dejó por última vez el estado de en ejecución, cargando los valores previos del contador de programa y de otros registros.

En definitiva, el cambio de proceso, que implica un cambio de contexto, requiere un esfuerzo considerablemente superior al de un cambio de contexto.

Procesos y Threads

El concepto de proceso engloba dos conceptos separados e independientes:

- *Unidad que posee recursos:* A un proceso se le asigna un espacio de memoria y se le puede asignar otros recursos, como dispositivos de E/S o ficheros.
- *Unidad a la que se le asigna el procesador:* Un proceso es un flujo de ejecución (una traza) a través de uno o más programas. Esta ejecución se entremezcla con la de otros procesos. La unidad planificada y expedida por el sistema operativo es el proceso.

Estas dos características son, de hecho, la esencia de un proceso. Sin embargo, son independientes, y pueden ser tratadas como tales por el sistema operativo. Esta distinción ha conducido en los sistemas operativos actuales a desarrollar la construcción conocida como **thread**, cuyas traducciones más frecuentes son **hilo**, **hebra** y, por error, **proceso ligero**. Si se tiene esta división de características, la unidad de asignación de la CPU se conoce como hilo, mientras que a la unidad que posee recursos se le llama proceso. Dentro de un proceso puede haber uno o más hilos de control, cada uno con:

1. Un estado de ejecución (en ejecución, listo, bloqueado).
2. Un contexto de procesador, que se salva cuando no esté ejecutándose.
3. Una pila de ejecución.
4. Algún almacenamiento estático para variables locales.
5. Acceso a la memoria y a los recursos de ese trabajo que comparte con los otros hilos.

Los beneficios clave de los hilos se derivan de las implicaciones del rendimiento: se tarda menos tiempo en crear un nuevo hilo de un proceso que ya existe, en terminarlo, y en hacer un cambio de contexto entre hilos de un mismo proceso. Al someter a un mismo proceso a varios flujos de ejecución se mantiene una única copia en memoria del código, y no varias.

Planificación de procesos

Se van a analizar todos los aspectos relacionados con el problema de cuándo asignar un procesador y a qué proceso. Se distinguirán tres niveles o tipos de planificación: a largo, medio y corto plazo.

Niveles de Planificación.

Se define el **scheduling** (planificación) como el conjunto de políticas y mecanismos construidos dentro del sistema operativo que gobiernan la forma de conseguir que los procesos a ejecutar lleguen a ejecutarse. El **scheduling** está asociado a las cuestiones de:

- cuándo introducir un nuevo proceso en el sistema.
- determinar el orden de ejecución de los procesos del sistema.

Y muy relacionado con la gestión de los recursos. Existen tres niveles de **scheduling**, son:

- Planificador de la CPU, o a corto plazo.
- Planificador a medio plazo.
- Planificador a largo plazo.



Planificador de la CPU, o a corto plazo:

La planificación de la CPU, en el sentido de conmutarla entre los distintos procesos, es una de las funciones del sistema operativo. Este despacho es llevado a cabo por un pequeño programa llamado **planificador a corto plazo** o **dispatcher** (*despachador*). La misión del *dispatcher* consiste en asignar la CPU a uno de los procesos listos del sistema, para ello sigue un determinado algoritmo. Para que el *dispatcher* conmute el procesador entre dos procesos es necesario realizar un cambio de proceso. Los eventos que pueden provocar la llamada al *dispatcher*, pueden ser:

1. El proceso en ejecución acaba su ejecución o no puede seguir ejecutándose (por una E/S, operación WAIT sobre un semáforo a cero, etc).
2. Un elemento del sistema operativo ordena el bloqueo del proceso en ejecución.
3. El proceso en ejecución agota su *quantum* de estancia en la CPU.
4. Un proceso pasa a estado listo.

Hay que tener en cuenta que cuanto menos se llame al *dispatcher* menos tiempo ocupa la CPU un programa del sistema operativo, y aumenta el rendimiento de los procesos del usuario, puesto que un cambio de proceso cada vez que se llama al *dispatcher* lleva bastante tiempo. De esta forma, si sólo se activa el *dispatcher* como consecuencia de los dos primeros eventos se estará haciendo un buen uso del procesador. Un buen ejemplo sería en sistemas por lotes, en los que los programas no son interactivos. Un ejemplo de la activación del *dispatcher* como consecuencia de los dos últimos eventos sería un sistema de tiempo compartido, pues un proceso que se dedicara a realizar cálculos, y no realizara E/S, monopolizaría el uso de la CPU. Los sistemas operativos en que los dos últimos eventos no provocan la activación del dispatcher muestran preferencia por el proceso en ejecución, si no ocurre esto se tiene más en cuenta el conjunto de todos los procesos.

Planificación a Medio Plazo:

El planificador a medio plazo es el encargado de regir las transiciones de procesos entre memoria principal y secundaria, actúa intentando maximizar la utilización de los recursos. Por ejemplo, transfiriendo siempre a memoria secundaria procesos bloqueados, o transfiriendo a memoria principal procesos suspendidos-listos en lugar de suspendidos-bloqueados. Esto es muy útil en los sistemas de multiprogramación y tiempo compartido cuando un número de procesos finitos residen en la memoria principal (la cual posee un tamaño limitado) y se da el caso de que todos los procesos en memoria están bloqueados, desperdiándose así la CPU, por lo que se intercambian los procesos enteros entre memoria principal y memoria secundaria, normalmente discos, con lo que se aumenta el número de procesos, y, la probabilidad de un mayor rendimiento de la CPU.



Planificación a largo plazo:

Este planificador está presente en algunos sistemas que admiten además de procesos interactivos trabajos por lotes. Usualmente, se les asigna una prioridad baja a los trabajos por lotes, utilizándose estos para mantener ocupados a los recursos del sistema durante periodos de baja actividad de los procesos interactivos. Normalmente, los trabajos por lotes realizan tareas rutinarias como el cálculo.

El objetivo primordial del planificador a largo plazo es el de dar al planificador de la CPU una mezcla equilibrada de trabajos, tales como los limitados por la CPU (utilizan mucho la CPU) o la E/S. Así, por ejemplo, cuando la utilización de la CPU es baja, el planificador puede admitir más trabajos para aumentar el número de procesos listos y cuando resulta que la utilización de la CPU y el tiempo de respuesta llega a ser alta, el planificador a largo plazo puede optar por reducir la frecuencia de admisión de trabajos.

Normalmente, se invoca al planificador a largo plazo siempre que un proceso termina, aún así, la frecuencia de invocación es mucho menor que la de los otros dos planificadores.

Objetivos y Criterios de Planificación

El principal objetivo de la planificación a corto plazo es repartir el tiempo del procesador de forma que se optimicen algunos puntos del comportamiento del sistema. Generalmente se fija un conjunto de criterios con los que evaluar las diversas estrategias de planificación. El criterio más empleado establece dos clasificaciones.

En primer lugar, se puede hacer una distinción entre los criterios orientados a los usuarios y los orientados al sistema. Los criterios orientados al usuario se refieren al comportamiento del sistema tal y como lo perciben los usuarios o los procesos. Uno de los parámetros es el tiempo de respuesta, que es el periodo de tiempo transcurrido desde que se emite una solicitud hasta que la respuesta aparece en la salida. Otros criterios están orientados al sistema, esto es, se centran en el uso efectivo y eficiente del procesador, por ejemplo la productividad, es decir, el ritmo con que terminan los procesos.

Otra forma de clasificación es considerar los criterios relativos al rendimiento del sistema y los que no lo son. Los criterios relativos al rendimiento son cuantitativos y, en general, pueden evaluarse o ser analizados fácilmente. Algunos ejemplos son el tiempo de respuesta y la productividad. Los criterios no relativos al rendimiento son, en cambio, cualitativos y no pueden ser evaluados fácilmente. Un ejemplo de estos criterios es la previsibilidad. Sería conveniente que el servicio ofrecido a los usuarios tenga las mismas características en todo momento, independientemente de la existencia de otros trabajos ejecutados por el sistema. En particular, una disciplina de planificación debe:



- Ser equitativa: debe tratar a todos los procesos de la misma forma y no aplazar indefinidamente a ningún proceso. La mejor es que mientras un proceso espera un recurso su prioridad crezca.
- Ser eficiente: debe maximizar el uso de los recursos tales como intentar que la ocupación de la CPU sea máxima y reducir el gasto extra de un trabajo no productivo.
- Tener un tiempo de respuesta bueno: para que los usuarios interactivos reciban respuesta en tiempos aceptables.
- Tener un tiempo de proceso global predecible: Un proceso debe ejecutarse aproximadamente en el mismo tiempo y casi al mismo costo con independencia de la carga del sistema.
- Aumentar al máximo la productividad o el rendimiento: maximizar el número de trabajos procesados por unidad de tiempo, dando preferencia a los procesos que ocupan recursos decisivos y favoreciendo a los procesos que muestran un comportamiento deseable. En el primer caso conseguimos liberar el recurso cuanto antes para que esté disponible para un proceso de mayor prioridad. Con el segundo criterio escogemos a los procesos que no consumen muchos recursos dejándole al sistema mayor capacidad de actuación.

Estos criterios son dependientes entre sí y es imposible optimizar todos de forma simultánea. Por ejemplo, obtener un buen tiempo de respuesta puede exigir un algoritmo de planificación que alterne entre los procesos con frecuencia, lo que incrementa la sobrecarga del sistema y reduce la productividad. Por tanto, en el diseño de un política de planificación entran en juego compromisos entre requisitos opuestos; el peso relativo que reciben los distintos requisitos dependerá de la naturaleza y empleo del sistema.

Planificación apropiativa y no apropiativa

Una disciplina de planificación es no apropiativa si una vez que la CPU ha sido asignada al proceso, ya no se le puede arrebatar. Y por el contrario, es apropiativa, si se le puede quitar la CPU. La planificación apropiativa es útil en los sistemas en los que los procesos de alta prioridad requieren una atención rápida. En los de tiempo real, por ejemplo, las consecuencias de perder una interrupción pueden ser desastrosas. En los sistemas de tiempo compartido, la planificación apropiativa es importante para garantizar tiempos de respuesta aceptables. La apropiación tiene un precio. El cambio de proceso implica gasto extra. Para que la técnica de apropiación sea efectiva deben mantenerse muchos procesos en memoria principal, de manera que el siguiente proceso se encuentre listo cuando quede disponible la CPU. Conservar en memoria principal procesos que no están en ejecución implica gasto extra. En los sistemas no apropiativos, los trabajos largos retrasan a los cortos, pero el tratamiento para todos los procesos es “más justo” (en el sentido de que una vez asignada la CPU a un proceso, no es desplazado por otro). Los tiempos de respuesta son más predecibles porque los trabajos nuevos de alta prioridad no pueden desplazar a los trabajos en espera. Al diseñar mecanismos de planificación apropiativa no hay que perder de vista la arbitrariedad de casi todos los sistemas de prioridades. Se puede construir un mecanismo complejo para implantar fielmente un esquema de

apropiación por prioridades sin que, de hecho, se hayan asignado prioridades de forma coherente.

El reloj de interrupciones

Un proceso se encuentra en ejecución cuando tiene asignada la CPU. Si el proceso pertenece al sistema operativo, se dice que el sistema operativo está en ejecución y que puede tomar decisiones que afectan al sistema. Para evitar que los usuarios monopolicen el sistema, de forma deliberada o accidentalmente, el sistema operativo tiene mecanismos para arrebatar la CPU al usuario.

El sistema operativo gestiona un reloj de interrupciones que genera una interrupción cada cierto tiempo. Un proceso mantiene el control de la CPU hasta que la libera voluntariamente (acaba su ejecución, o se bloquea), hasta que el reloj interrumpe o hasta que alguna otra interrupción desvía la atención de la CPU. Si el usuario se encuentra en ejecución y el reloj interrumpe, el sistema operativo entra en ejecución para comprobar, por ejemplo, si ha pasado el cuanto de tiempo del proceso que estaba en ejecución. El reloj de interrupciones asegura que ningún proceso acapare la utilización del procesador. El sistema operativo, apoyándose en él, intenta distribuir el tiempo de CPU entre los distintos procesos ya sean de E/S o de cálculo. Por tanto, ayuda a garantizar tiempos de respuesta para los usuarios interactivos, evitando que el sistema quede bloqueado en un ciclo infinito de algún usuario, y permite que los procesos respondan a eventos dependientes del tiempo. Los procesos que deben ejecutarse periódicamente dependen del reloj de interrupciones. No se debe confundir en ningún caso al reloj de interrupciones con el reloj de la máquina o reloj hardware.

Planificación con Prioridades

La mayoría de los algoritmos de planificación apropiativos emplean el uso de prioridades de acuerdo con algún criterio. Cada proceso tiene una prioridad asignada y el planificador seleccionará siempre un proceso de mayor prioridad antes que otro de menor prioridad. Las prioridades pueden ser asignadas de forma automática por el sistema, o bien se pueden asignar externamente. Pueden ganarse o comprarse. Pueden ser estáticas o dinámicas. Pueden asignarse de forma racional, o de manera arbitraria en situaciones en las que un mecanismo del sistema necesita distinguir entre procesos, pero no le importa cuál de ellos es en verdad más importante. Las prioridades estáticas no cambian. Los mecanismos de prioridad estática son fáciles de llevar a la práctica e implican un gasto extra relativamente bajo. Sin embargo, no responden a cambios en el entorno, que podrían hacer necesario un ajuste de prioridades. Las prioridades dinámicas responden a los cambios. La prioridad inicial asignada a un proceso tiene una corta duración, después se ajusta a un valor más apropiado, a veces deducido de su comportamiento. Los esquemas de prioridad dinámica son más complejos e implican un mayor gasto extra que puede quedar justificado por el aumento en la sensibilidad del sistema.



Algoritmos de planificación

Para evaluar los algoritmos que se verán a continuación hay que tener en cuenta ciertas medidas que se utilizan para evaluarlos.

- a. Porcentaje de utilización de la CPU por procesos de usuario: La CPU es un recurso caro que necesita ser explotado, los valores reales suelen estar entre un 40% y un 90%.
- b. Rendimiento (*throughput*) = número de ráfagas por unidad de tiempo. Se define una *ráfaga* como el periodo de tiempo en que un proceso necesita la CPU; un proceso, durante su vida, alterna ráfagas con bloqueos. Por extensión, también se define como el número de trabajos por unidad de tiempo.
- c. Tiempo de espera (E) = tiempo que una ráfaga ha permanecido en estado listo.
- d. Tiempo de finalización (F) = tiempo transcurrido desde que una ráfaga comienza a existir hasta que finaliza.
 $F = E + t$ (t = tiempo de CPU de la ráfaga).
- e. Penalización (P) = $(E + t) / t = F / t$, es una medida adimensional que se puede aplicar homogéneamente a las ráfagas independientemente de su longitud.

En general, hay que maximizar los dos primeros parámetros y minimizar los tres últimos. Sin embargo, estos objetivos son contradictorios, el dedicar más tiempo de CPU a los usuarios se hace a costa de llamar menos al algoritmo de planificación (menos cambios de proceso) y de simplificarlo. Esto provoca que la CPU se reparta menos equitativamente entre los procesos, en detrimento de los últimos tres parámetros. Así pues, dependiendo de los objetivos se elegirá cierto algoritmo. En los sistemas por lotes suele primar el rendimiento del sistema, mientras que en los sistemas interactivos es preferible minimizar, por ejemplo, el tiempo de espera. Para más información sobre los algoritmos de planificación consultar el anexo del mismo nombre.



1.3.6 Gestión De Memoria.

Conceptos generales

Para que un proceso pueda ejecutarse debe estar ubicado en la memoria principal del ordenador. Una parte del sistema operativo se va a encargar de gestionar la memoria principal, de forma que los procesos puedan residir en la memoria sin conflictos. La gestión de la memoria implica varias tareas, una de ellas es llevar un registro de qué zonas están libres (es decir, no están siendo utilizadas por ningún proceso), y qué zonas están ocupadas, y por qué procesos.

Otra tarea importante surge en sistemas en los que no todos los procesos, o no todo el código y datos de un proceso, se ubican en la memoria principal. En estos sistemas, a menudo se debe pasar parte, o la totalidad del código y datos de un proceso, de memoria a disco, o viceversa; siendo el sistema operativo responsable de esta tarea. De esta forma se libera al usuario de realizar estas transferencias de información, de las cuales no es consciente.

También es importante en la gestión de la memoria la cuestión de la carga de los programas de disco a memoria y el de la protección y el problema de la protección, que surge en el momento en que varios procesos deben compartir la memoria del ordenador. En general, se pretende que un proceso no pueda modificar las direcciones de memoria en las que no reside. Esto es así porque en las direcciones de memoria donde no está ubicado el proceso pueden residir otros procesos, o código y/o estructuras de datos del sistema operativo. Si un proceso puede modificar indiscriminadamente la memoria, podría, por ejemplo, cambiar el valor de una dirección de memoria donde residiera una variable de otro proceso, con la consecuente ejecución incorrecta del proceso propietario de la variable. Algunos sistemas ni siquiera permiten que un proceso pueda leer las direcciones de memoria en las que no reside, con esto se consigue privacidad sobre el código y datos de los procesos.

Existen varias formas de gestionar la memoria. Por lo común, la forma de gestión dependerá de la máquina virtual que se quiera proporcionar y del *hardware* subyacente. Con independencia de la forma de gestión es necesario decidir qué estrategias se deben utilizar para obtener un rendimiento óptimo. Las estrategias de administración de la memoria especifican el comportamiento de una organización de memoria determinada cuando se siguen diferentes políticas.

Los sistemas actuales son en su mayor parte sistemas con almacenamiento virtual. Muchas de la formas de gestión estudiadas a continuación tienen principalmente valor histórico, pero sientan las bases de los sistemas actuales.



Jerarquía de la memoria:

Los programas y datos necesitan estar en la memoria principal para ser ejecutados, o para poder ser referenciados. Los programas o datos que no se necesitan de inmediato pueden guardarse en la memoria secundaria hasta que se necesiten, y en ese momento se transfieren a la memoria principal para ser ejecutados o referenciados. Los soportes de memoria secundaria, como cintas o discos, son en general menos caros que la memoria principal, y su capacidad es mucho mayor. Normalmente, es mucho más rápido el acceso a la memoria principal que a la secundaria.

En los sistemas con varios niveles de memoria hay muchas transferencias constantes de programas y datos entre los distintos niveles. Éstas consumen recursos del sistema, como tiempo de la CPU, que de otro modo podrían utilizarse provechosamente. La jerarquía de la memoria se extiende un nivel más, con una clara mejora del rendimiento. Este nivel adicional, la *memoria caché*, es una memoria de alta velocidad, mucho más rápida que la memoria principal. La memoria caché es extremadamente cara, si se compara con la principal, por lo que sólo se utilizan memorias caché relativamente pequeñas. La Figura 7 muestra la relación que existe entre la memoria caché, la principal y la secundaria.

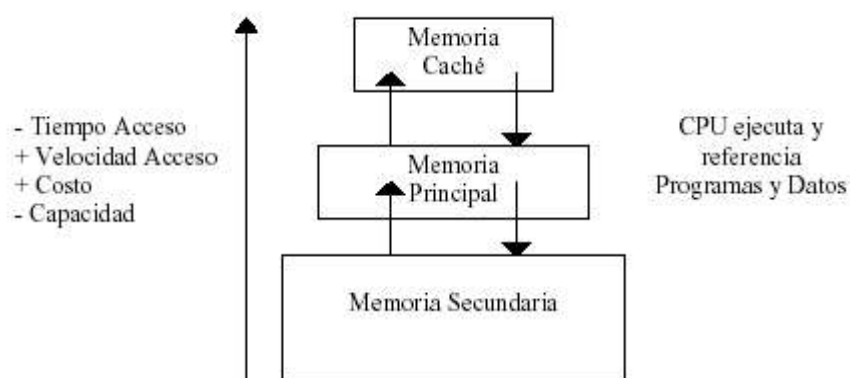


Figura 7. Jerarquía de Memoria.

La memoria caché introduce un nivel adicional de transferencia de información en el sistema. Los programas en memoria principal se pasan a la memoria caché antes de ejecutarse. En la memoria caché se pueden ejecutar mucho más rápido que en la principal. El objetivo es que el trabajo extra requerido por la transferencia de programas sea mucho menor que el incremento del rendimiento obtenido por la ejecución más rápida en la caché.

Gestión de la memoria en los sistemas monoprogramados

En los sistemas de monoprogramación sólo existe un proceso de usuario, que disfruta de todos los recursos del ordenador. Esto va a simplificar notablemente la gestión de la memoria, ya que ésta sólo debe ser compartida por los programas del sistema operativo, y por el único proceso de usuario existente.

Dependiendo del tipo de diseño, el sistema operativo ocupará:

1. La parte baja de la memoria RAM.
2. La parte alta de la memoria ROM.
3. IBM ubica parte del sistema operativo en RAM, y los gestores de dispositivos en ROM, a esta última parte se le llama BIOS (*Basic Input/Output System*, sistema básico de entrada/salida).

Si el usuario conoce la ubicación en la memoria del sistema operativo, entonces puede escribir programas en términos de direcciones absolutas de memoria. Una *dirección absoluta* de memoria es una dirección física ó real de la memoria. En contraposición se tienen las direcciones relativas. Un programa está escrito en término de *direcciones relativas* suponiendo que empieza a cargarse en la dirección cero de la memoria. Por lo general, los usuarios escriben programas en lenguajes de alto nivel, por lo que son los traductores los encargados de generar las direcciones que ocupan las variables, procedimientos, etc, en la memoria. Los compiladores no generan direcciones absolutas de memoria, pues no saben dónde se almacenarán los procesos.

Por lo común, los sistemas operativos monousuario de monoprogamación (muy comunes en las microcomputadoras) no tienen protección de la memoria. Por lo tanto, el único proceso de usuario que existe en la memoria, puede modificar posiciones de memoria pertenecientes al sistema operativo, esto provocaría errores al ejecutarse la zona modificada. La protección se puede realizar mediante un *registro de límite* integrado en la CPU. Si se tiene un esquema de diseño como el de la opción (b) anterior, el registro de límite contendrá la dirección de inicio de carga del sistema operativo. El *hardware*, en tiempo de ejecución, verifica que las direcciones generadas por el proceso de usuario no son superiores al valor del registro de límite. En caso de ser superior, el proceso de usuario intenta acceder al sistema operativo, esto provoca una interrupción *hardware* que gestiona el sistema operativo, normalmente eliminando al proceso.

Gestión de la memoria en los sistemas multiprogramados

En un sistema de multiprogamación la memoria debe ser compartida por varios procesos de cara a obtener una mayor utilización de los recursos del ordenador. Esto provoca que la gestión de la memoria se complique substancialmente. En primer lugar, hay que llevar un recuento de las zonas de memoria ocupadas por los procesos. Así, cuando un nuevo proceso entre en la memoria se le asignará una zona que estaba libre. Otro problema a resolver viene dado por el hecho de que en el momento de escribir un programa no se sabe en qué zona de memoria se ubicará, siendo posible que durante la vida de un proceso éste cambie varias veces de emplazamiento. Habrá que tener en cuenta, también, la protección de las zonas de memoria ocupadas por los procesos, máxime en sistemas multiusuario, donde los procesos pueden pertenecer a distintos usuarios.

A continuación se estudiará varias tipos de gestión de la memoria utilizadas en sistemas de multiprogamación.

Asignación de memoria contigua:

En un esquema de asignación de memoria contigua un proceso se ubica en su totalidad en posiciones consecutivas de memoria. Un ejemplo de este tipo de asignación es el utilizado en los sistemas de monoprogramación vistos previamente.

Particiones estáticas:

Esta forma de gestión consiste en dividir la memoria en varias zonas, pudiendo ser cada zona de un tamaño diferente (ver la Figura 8). El tamaño de las zonas podrá ser modificado eventualmente por algún usuario responsable de la administración del ordenador. Los trabajos se traducían mediante compiladores y ensambladores absolutos, para ejecutarse en una partición específica. Una vez introducido un proceso en una partición, permanece en ella hasta su finalización. Si un trabajo se iniciaba, y la partición para la que estaba compilado estaba ocupada, tenía que esperar, aunque estuvieran libres otras particiones. Esto provocaba una pérdida de eficiencia.

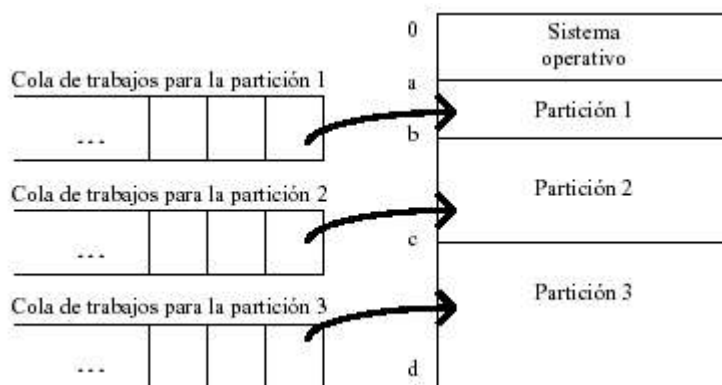


Figura 8. Multiprogramación con particiones estáticas, con traducción y carga absolutas.

Para mejorar el rendimiento es preciso que un proceso se pueda cargar en cualquier partición. Para ello, los programas se escriben en términos de direcciones relativas a la dirección de comienzo de carga cero. Sin embargo, los programas no se cargan a partir de la dirección cero, esta circunstancia se debe resolver mediante la *reasignación* o *reubicación*. Una posible implantación de esta viene proporcionada por el *hardware*. Existe un registro, denominado *registro base*, en el que el sistema operativo, dentro de la operación de cambio de proceso, escribe la dirección de memoria a partir de la cual se almacenó el proceso. Esta dirección coincidirá con la de comienzo de la partición en que reside, y forma parte de su descriptor. Cuando la CPU genera una dirección de memoria, ésta es transformada por el *hardware* antes de ser introducida en el bus del sistema. La transformación consiste en sumarle a la dirección el registro base. Para aclarar esto, supóngase que la instrucción que actualmente ejecuta la CPU guarda el contenido del acumulador en la dirección relativa 100 de la memoria. Esta dirección, 100, (que podría, por ejemplo, guardar el contenido de una variable de otro proceso) es relativa a una dirección 0 de comienzo del programa. Si el programa se

ha cargado en la posición 10000 de la memoria, el registro base contendrá el valor 10000. Cuando la CPU ejecuta la instrucción, genera la dirección 100 de memoria (la cual físicamente pertenece a otro proceso). Sin embargo, el *hardware* suma 10000 a este valor, introduciéndose 10100 en el bus del sistema, dirección que realmente ocupa la variable. Con este esquema, si se quiere reubicar el proceso a otro lugar de la memoria, sólo hay que desplazarlo de lugar, y modificar el registro base con la nueva dirección de comienzo de carga.

Una solución *software* al problema de la reasignación consiste en modificar las instrucciones cuando el programa se carga en la memoria. Para que esto ocurra es preciso que el enlazador (el programa que a partir de los ficheros objeto genera un único objeto) incluya en el programa binario una lista que indique qué partes del programa son direcciones a reasignar, y cuáles no (constantes, códigos de operadores u otros elementos que no deban ser reasignados). Con esta solución, cada reubicación en la memoria implica una reasignación de las direcciones de memoria del programa.

Si se tiene el esquema *hardware* del registro base, para lograr la protección de las zonas de memoria basta con añadir un nuevo registro, denominado *registro límite*. Este registro guarda la última dirección de la partición, y forma también parte del PCB del proceso. El *hardware*, después de sumar el registro base a la dirección relativa, comprueba que la dirección obtenida no supere el valor del registro límite. Si se supera el valor, se está intentando acceder a una zona que no corresponde al proceso; en esta situación, el *hardware* genera una interrupción. El sistema operativo sirve a la interrupción, lo normal es que mande una señal al proceso por violación de memoria. Si el proceso no tiene definido atrapar esa señal, lo cual es lo más probable, se eliminará al proceso.

Una última observación, al margen de la protección: cuando un proceso se introduce en una partición, lo más probable es que su tamaño no sea el mismo (es decir, sea algo menor) que el de la partición. Esto origina un problema de desperdicio de memoria conocido como *fragmentación interna*.

Particiones dinámicas:

En este método se va asignando la memoria dinámicamente a los procesos, conforme se introducen en la memoria. A cada proceso se le asigna exactamente la memoria que necesita.

En la Figura 9 se ilustra cómo evoluciona la ocupación de la memoria en un sistema de este tipo. Al principio sólo se encuentra el proceso A en la memoria. Después, se insertan los procesos B y C. En la Figura 9-(d) A concluye. Luego, D entra y B sale. Por último E entra.

Con este método de gestión de la memoria se evita el problema de la fragmentación interna. Sin embargo, aparece el problema de la *fragmentación externa* entre particiones, el cual se aprecia en la Figura 9-(f). El problema consiste en que se creen huecos libres demasiado pequeños como para que quepan procesos, aunque la unión de todos esos huecos produciría un hueco considerable, lo que acarrea el desperdicio de la memoria. Una posible solución es la *compactación* de la memoria, que consiste en desplazar todos los procesos hacia la parte inferior de la memoria mientras sea posible. Como la compactación lleva mucho tiempo, a veces no se realiza, o se hace por la noche, en

horas de poco uso del ordenador. Hay que tener en cuenta que el sistema debe detener todas sus actividades mientras realiza la compactación. Ello puede ocasionar tiempos de respuesta irregulares para usuarios interactivos, y podría ser devastador en un sistema de tiempo real. Además, con una combinación normal de trabajos que cambia rápidamente, es necesario compactar a menudo. En este caso, los recursos del sistema que se consumen quizá no justifiquen las ventajas de la compactación.

El esquema de los registros base y límite sigue siendo válido para la reasignación y la protección. Otro tema a tener en cuenta es la cantidad de memoria por asignar a un proceso recién creado. Si los procesos se crean con un tamaño fijo invariante, la asignación es muy sencilla, se asigna exactamente lo que se necesite.

Si, por el contrario, los segmentos de datos de los procesos pueden crecer, como es el caso de la asignación dinámica de memoria a partir de una pila, que ocurre en muchos lenguajes de programación, aparece un problema cuando un proceso intenta crecer. Si hay un hueco adyacente al proceso, éste puede ser asignado, y el proceso podrá crecer hacia el hueco. Sin embargo, si el proceso es adyacente a otro proceso, el proceso de crecimiento deberá ser desplazado a un hueco de la memoria lo suficientemente grande; o bien, habrá que eliminarlo.

Si es de esperar que la mayoría de los procesos crezcan conforme se ejecuten, sería una buena idea asignar un poco de memoria adicional siempre que un proceso pase a la memoria, con el fin de reducir el gasto excesivo asociado con el traslado de procesos que ya no tienen espacio suficiente en su memoria asignada. En la Figura 10-(a) vemos una configuración de la memoria en la que se asignó a dos procesos el espacio adicional para el crecimiento. Si los procesos pueden tener dos segmentos de crecimiento, como por ejemplo, el segmento de datos, que se utiliza como una pila, y el *stack*, se sugiere un método alternativo, el de la Figura 10-(b). En esta figura se puede ver que cada proceso tiene un *stack* de crecimiento hacia abajo, en la parte superior de la memoria asignada a él; además, tiene un segmento de datos justo encima del programa, el cual crece hacia arriba. La memoria entre ellos se puede utilizar para cualquiera de los segmentos. Si el espacio se agota, puede ocurrir que el proceso sea desplazado a un hueco con el espacio suficiente; o bien, ser aniquilado.

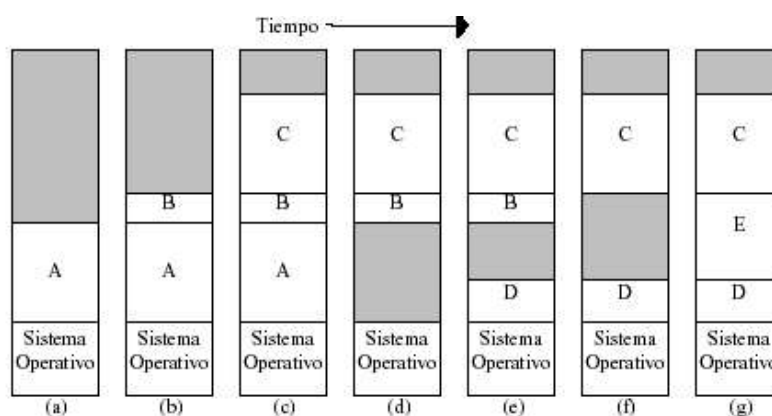


Figura 9. Asignación de memoria del proceso en el tiempo. Regiones sombreadas son memoria libre.

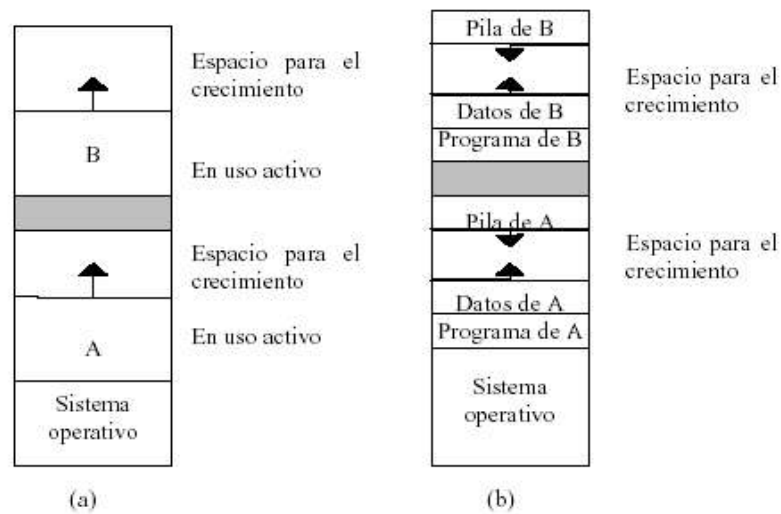


Figura 10. Asignación de espacio: (a) segmento de datos que crece, (b) pila y segmento de datos que crece.

Registro de la ocupación de la memoria:

En el sistema de particiones estáticas es sencillo llevar el registro de la ocupación de la memoria, basta con guardar sobre cada partición si está libre, u ocupada y por qué proceso, así como sus direcciones de comienzo y fin de partición. Por contra, con las particiones dinámicas, el número de éstas varía con el tiempo, así como su tamaño. Una forma posible de registrar la ocupación de la memoria es utilizar una lista enlazada de los segmentos de la memoria asignados o libres. La memoria de la Figura 11-(a) se presenta en la Figura 11-(c) como una lista enlazada de segmentos. Cada entrada de la lista especifica un hueco (H) o un proceso (P), la dirección donde comienza, su longitud, y un puntero a la siguiente entrada.

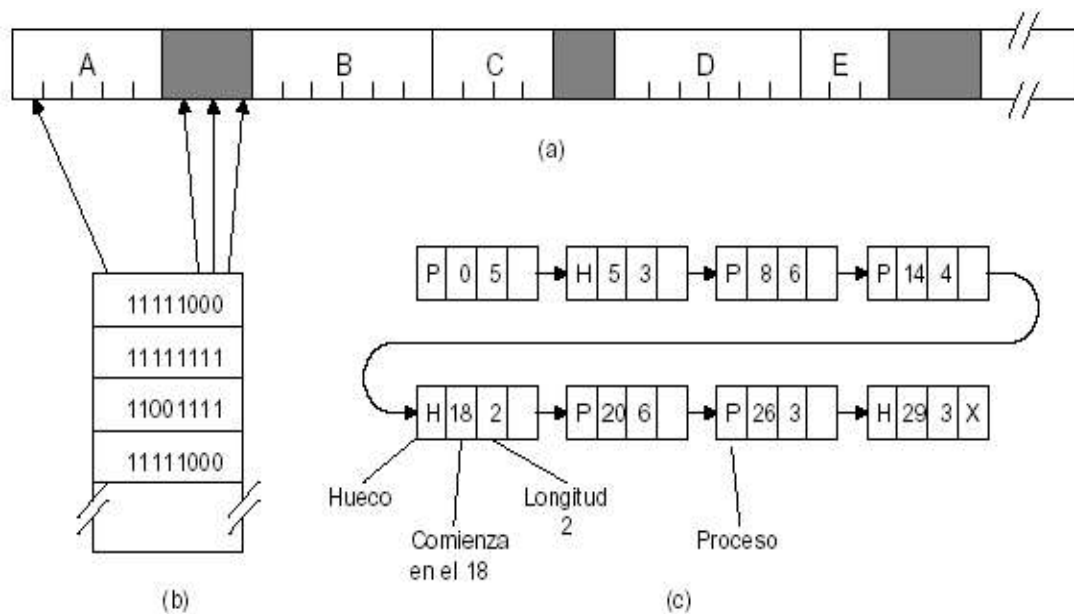


Figura 11. (a) Una parte de la memoria, con 5 procesos y 3 huecos. La marca muestra las unidades de asignación de la memoria. Las regiones sombreadas están libres. (b) El mapa de bits correspondiente. (c) Igual pero como lista enlazada.

Estrategias de colocación:

Cuando en un sistema de particiones dinámicas se debe asignar memoria principal para un nuevo proceso, y los procesos y huecos se mantienen en una lista ordenada por direcciones, se pueden utilizar diversos algoritmos para la elección del hueco de memoria donde ubicar al proceso. Supongamos que se conoce la cantidad de memoria por asignar.

El algoritmo más simple es el *primero en ajustarse (first fit)*. Se revisa la lista de huecos hasta encontrar un espacio lo suficientemente grande. El espacio se divide entonces en dos partes, una para el proceso, y otra para la memoria no utilizada, excepto en el caso poco probable de un ajuste perfecto. Este algoritmo es rápido, ya que busca lo menos posible.

Otro algoritmo es el *mejor en ajustarse (best fit)*, el cual busca en toda la lista, y elige el mínimo hueco suficientemente grande como para ubicar al proceso. Este algoritmo intenta que los huecos que se creen en la memoria sean lo más pequeños posible.

Como ejemplo de los algoritmos, retomemos la Figura 11. Si se necesita un bloque de tamaño 2, el primero en ajustarse asignará el espacio en 5, mientras que el mejor en ajustarse asignará el espacio en 18.

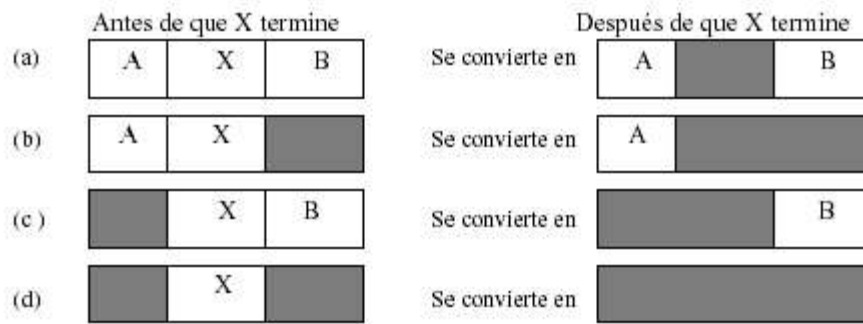


Figura 12. Cuatro combinaciones de vecinos para el proceso X que concluye.

Realizando simulaciones se ha demostrado que el algoritmo del mejor ajuste desperdicia más la memoria, pues al crear huecos demasiado pequeños, éstos no pueden ser utilizados por procesos. Un algoritmo que enfrenta el problema de la manera contraria es el del *peor ajuste (worst fit)*. En este algoritmo se elige el hueco más grande disponible. De esta forma aumenta la probabilidad de que el nuevo hueco creado sea lo suficientemente grande como para albergar un proceso.

Intercambio (swapping):

En un sistema con particiones estáticas el número de procesos con posibilidades de estar en estado listo viene determinado por el número de particiones, y en uno de particiones dinámicas por el tamaño de la memoria principal y el tamaño de los procesos, ya que en ambos métodos un proceso permanece en una partición hasta que finaliza.

Supongamos un sistema en que dicho número es cinco, es muy probable que en un momento dado los cinco procesos que ocupan las particiones estén bloqueados (por ejemplo, porque esperan la finalización de una operación de E/S). Mientras los cinco procesos permanezcan bloqueados se desperdicia la CPU. Para remediar este inconveniente muchos sistemas operativos optaron por permitir ejecutar concurrentemente más procesos de los que caben físicamente en la memoria principal del ordenador. Para ello se utiliza la memoria secundaria (generalmente los discos) que, aunque más lenta, tiene mayor capacidad de almacenamiento que la principal.

La solución consiste en tener en disco una copia de la parte de la memoria que ocupa todo proceso. En el disco se encuentran todos los procesos, en la memoria sólo unos cuantos. Para que un proceso se pueda ejecutar debe residir en memoria principal. La razón por la que se aumenta el número de procesos con posibilidades de tener instrucciones en memoria principal es porque cuanto mayor sea este número, es menos probable que se dé la circunstancia de que todos estén bloqueados y, por lo tanto, es menor la posibilidad de que la CPU permanezca inactiva.

Asignación de memoria no contigua:

Hasta ahora se han estudiado esquemas de administración de la memoria en los que los procesos se almacenan en posiciones contiguas (consecutivas) de memoria. Sin embargo, un proceso puede dividirse en bloques, y estos bloques pueden situarse en posiciones no contiguas de memoria principal. Es más, no es preciso que se encuentren en la memoria todos los bloques de un proceso para que se pueda ejecutar, basta con que se encuentren los bloques que contienen código o datos actualmente referenciados, el resto puede permanecer en memoria secundaria.

La memoria virtual:

La clave del concepto de memoria virtual es la disociación de las direcciones a las que hace referencia un proceso en ejecución de las direcciones disponibles en la memoria principal. Las direcciones a las que hace referencia un proceso en ejecución, en este esquema, se llaman *direcciones virtuales*. El intervalo de direcciones virtuales a las que puede hacer referencia un proceso en ejecución se llama *espacio de direcciones virtuales*, V , del proceso. El intervalo de direcciones reales de la memoria principal de un ordenador concreto se llama *espacio de direcciones reales*, R . El número de direcciones de V se denota $|V|$, y el número de direcciones de R , $|R|$. En los sistemas de almacenamiento virtual ya implantados lo normal es que $|V| \gg |R|$, aunque se han construido sistemas en los que $|V| < |R|$.

La memoria virtual es una técnica de gestión de la memoria que posibilita que el espacio de direcciones virtuales sea mayor al espacio de direcciones reales. En otras palabras, se permite hacer programas de tamaño mayor al de la memoria principal. Para lograr esto, el sistema operativo se encarga de mantener en la memoria principal solamente aquellas partes del espacio de direcciones del proceso que actualmente están siendo referenciadas, el resto permanece en disco.

La memoria virtual se basa en el hecho de que muchos programas presentan un comportamiento conocido como *operación en contexto o localidad*, según el cual, en cualquier intervalo pequeño de tiempo un programa tiende a operar dentro de un módulo lógico en particular, sacando sus instrucciones de una sola rutina y sus datos de una sola zona de datos. De esta forma, las referencias a memoria de los programas tienden a agruparse en pequeñas zonas del espacio de direcciones. La localidad de estas referencias viene reforzada por la frecuente existencia de bucles: cuanto más pequeño sea el bucle, menor será la dispersión de las referencias.

La memoria virtual se compagina con la multiprogramación. Al no tener que almacenar los procesos enteros en la memoria, caben más en la memoria principal, con lo que es más probable que siempre exista un proceso en estado listo. Por otro lado, cuando un proceso espera a que se cargue en la memoria principal parte de su código o datos, se inicia una E/S con el disco. Mientras dura dicha E/S la CPU puede ejecutar otro proceso.

Aunque los procesos sólo hacen referencia a direcciones virtuales, deben ejecutarse en la memoria real. Por lo tanto, durante la ejecución de un proceso es preciso establecer la correspondencia entre las direcciones virtuales y las reales.

Se han desarrollado varios métodos para asociar las direcciones virtuales con las reales. Los mecanismos de *traducción dinámica de direcciones* convierten las direcciones virtuales en direcciones reales en tiempo de ejecución. Todos estos sistemas tienen la propiedad de que las direcciones contiguas en el espacio de direcciones virtuales de un proceso no son necesariamente contiguas en la memoria principal: *contigüidad artificial*.

Esquema general de traducción:

Los mecanismos de traducción dinámica de direcciones deben mantener *mapas de correspondencia de traducción de direcciones* que indiquen qué localidades de la memoria virtual están en memoria principal en un momento dado y dónde se encuentran.

Existen dudas en cuanto a si los bloques en que se dividen los procesos deben ser del mismo tamaño o de tamaños diferentes. Cuando los bloques son del mismo tamaño, se llaman **páginas**, y la organización de la memoria virtual correspondiente se conoce como **paginación**. Cuando los bloques pueden tener tamaños diferentes se llaman **segmentos**, y la organización de la memoria virtual correspondiente se llama **segmentación**. Algunos sistemas combinan ambas técnicas, con segmentos, que son entidades de tamaño variable, compuestos de páginas de tamaño fijo. Las direcciones en un sistema de correspondencia son bidimensionales. Para referirse a un elemento en particular, el programa especifica el bloque en el que se encuentra el elemento, y su desplazamiento a partir del inicio del bloque. Una dirección virtual, v , se denota por un par ordenado (b,d) , donde b es el número de bloque en el que se encuentra el elemento al que se hace referencia, y d es el desplazamiento a partir del inicio del bloque.

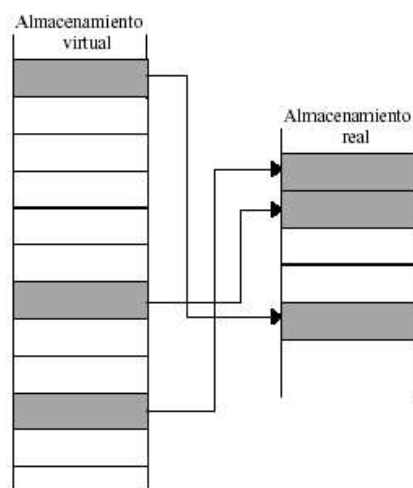


Figura 13. Correspondencia entre elementos del espacio de direcciones virtuales y reales.

La traducción de una dirección virtual $v = (b,d)$ a una dirección real, r , se lleva a cabo de la siguiente forma (Figura 14). Cada proceso tiene su propia *tabla de correspondencia de bloques*, mantenida por el sistema operativo dentro de la memoria principal. Un registro especial dentro de la CPU, denominado *registro de origen de la tabla de*

correspondencia de bloques, se carga con la dirección real, a , de la tabla de correspondencia de bloques del proceso durante el cambio de proceso. La tabla contiene una entrada por cada bloque del proceso, y las entradas siguen un orden secuencial para el bloque 0, el bloque 1, etcétera. Ahora se suma el número de bloque, b , a la dirección base, a , de la tabla de bloques, para formar la dirección real de la entrada del bloque b en la tabla de correspondencia de bloques. Esta entrada contiene la dirección real, b' , de inicio del bloque b . El desplazamiento, d , se suma a la dirección de inicio del bloque, b' , para formar la dirección real deseada, $r = b' + d$.

Es importante señalar que la traducción de una dirección virtual a real la realiza una unidad *hardware*, que transforma todas las direcciones generadas por la CPU antes de que pasen al bus del sistema. Es esencial que esta transformación la realice el *hardware*, y no el sistema operativo, pues muchas instrucciones máquina incluyen referencias a memoria, y la correspondencia debe realizarse rápidamente, para no ralentizar en exceso el tiempo de ejecución de los procesos. Por ejemplo, las dos sumas indicadas en la Figura 14 deben ser más rápidas que las sumas convencionales del lenguaje máquina.

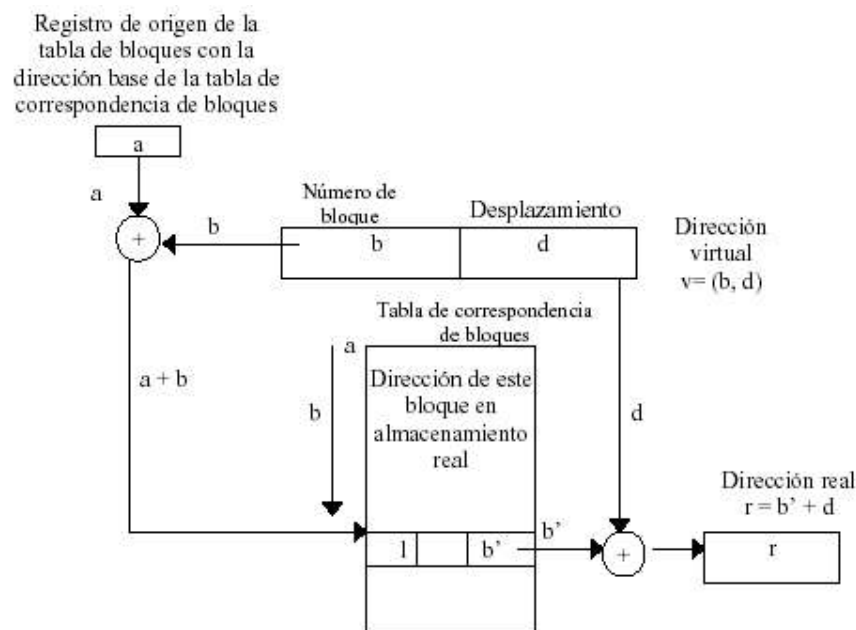


Figura 14. Traducción de direcciones virtuales con correspondencia de bloques.

Aunque el *hardware* consulta las tablas de correspondencia de bloques para la transformación de direcciones, es el sistema operativo el encargado de rellenar y gestionar dichas tablas. Un proceso no tiene por qué tener todos sus bloques en memoria principal, recuérdese que el espacio de direcciones virtuales puede ser muy superior al espacio de direcciones reales, esto hace que a veces un proceso referencie a una dirección de un bloque que no se encuentra en la memoria principal. Para detectar esto, las tablas de correspondencias tienen un "bit de presencia" por entrada (cada entrada representa un bloque), que indica si el bloque se encuentra presente en la memoria principal o no. El

hardware de traducción debe verificar este bit en cada referencia a memoria. Si el bloque no está en memoria principal, el *hardware* produce una interrupción. Esta interrupción provoca que el control pase al *software* (sistema operativo), para que éste inicie la transferencia del bloque que falta desde la memoria secundaria a la memoria principal, y actualice de acuerdo con ello la tabla de correspondencias. El proceso en ejecución se hará no listo hasta que se haya completado esta transferencia. La posición de los bloques en la memoria secundaria puede guardarse en la misma tabla de correspondencias.

Paginación:

El almacenamiento a un sólo nivel puede llevarse a cabo mediante una técnica llamada *paginación*, según la cual el espacio de direcciones virtuales se divide en *páginas* del mismo tamaño. La memoria principal se divide también en *marcos* o *páginas físicas* del mismo tamaño. Estos marcos son compartidos entre los distintos procesos que haya en el sistema, de forma que en cualquier momento un proceso dado tendrá unas cuantas páginas residentes en la memoria principal (sus *páginas activas*) y el resto en la memoria secundaria (sus *páginas inactivas*). El mecanismo de paginación cumple dos funciones:

1. llevar a cabo la transformación de una dirección virtual a física, o sea, la determinación de la página a la que corresponde una determinada dirección de un programa, así como del marco, si lo hay, que ocupa esta página.
2. transferir, cuando haga falta, páginas de la memoria secundaria a la memoria principal, y de la memoria principal a la memoria secundaria cuando ya no sean necesarias.

Con el fin de determinar la página a la que hace referencia un programa, los bits de mayor peso de la dirección se interpretan como el número de página, y los bits de menor peso como el número de palabra dentro de esta página. De ahí que si el tamaño de página es 2^n , los n bits finales de la dirección representarán el número de palabra y los bits restantes del principio el número de página. El número total de bits en la dirección es suficiente para direccionar la totalidad de la memoria virtual. Así, por ejemplo, en el Atlas las direcciones de programa tenían 20 bits de longitud, proporcionando una memoria virtual de 220 palabras; el tamaño de la página era de 512 palabras (29), y de ahí que los 9 bits inferiores representasen el número de palabra y los 11 superiores el número de la página. El número total de páginas en la memoria virtual era por tanto de 211 (en contraposición a las 32 páginas físicas de que disponía la memoria principal).

La transformación de número de página y de palabra en la dirección física de memoria se realiza a través de una *tabla de páginas*, cuyo p -ésimo elemento contiene la posición p' del marco que contiene a la página p (la posibilidad de que la p -ésima página no se encuentre en la memoria principal se abordará dentro de un momento). El número de palabra, w , se suma a p' para obtener la dirección buscada (ver la Figura 15).



La transformación de direcciones consiste, pues, en:

$$f(a) = f(p, w) = p' + w$$

donde la dirección de programa, a , el número de página, p , y el número de palabra, w , están relacionados con el tamaño de página Z a través de:

$$p = \text{parte entera de } (a/Z) \quad w = \text{resto de } (a/Z)$$

Cada dirección lógica es transformada en alguna dirección física por el *hardware* de paginación. Observe también que si el tamaño de página (como es lo usual) es una potencia de dos, el *hardware* no precisa realizar ninguna división, simplemente sabe que los últimos n bits, si el tamaño de página es de 2^n , representan el desplazamiento, y los primeros bits la página.

Cada proceso debe tener su propia tabla de páginas, y su dirección de comienzo en la memoria principal forma parte de la porción del PCB utilizada para realizar un cambio de proceso.

Como el número de marcos (cantidad de memoria real) asignados a un proceso será normalmente menor que el número de páginas que éste utiliza, es muy posible que una dirección del programa haga referencia a una página que no se encuentre en aquel momento en la memoria principal. En este caso el elemento correspondiente de la tabla de páginas estará vacío, provocando el *hardware* una interrupción de "fallo de página" si se intenta acceder a ella. Esta interrupción provoca que el control pase al *software* (al sistema operativo), para que éste inicie la transferencia de la página que falta desde la memoria secundaria a la memoria principal, y actualice de acuerdo con ello la tabla de páginas.

Si no existe ningún marco vacío en el momento en que ocurre un fallo de página, hay que guardar en la memoria secundaria alguna otra página con el fin de hacer sitio a la nueva. La elección de la página que habrá que sacar es el resultado de un *algoritmo de reemplazo de página*, del cual veremos varios ejemplos más adelante. Por el momento, vamos a destacar tan sólo el hecho de que la información que necesita el algoritmo de cambio de página, puede estar contenida en algunos bits adicionales que se añaden a cada elemento de la tabla de páginas.



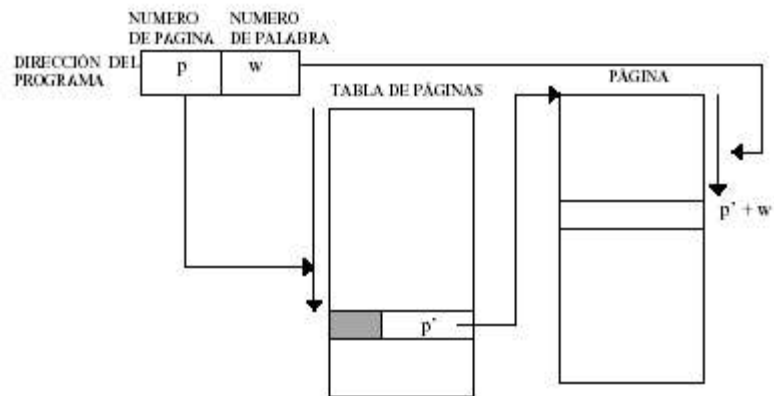


Figura 15. Transformación de direcciones para llevar a cabo la paginación.

Memoria asociativa:

En el sistema que hemos descrito el tiempo necesario para cada referencia a memoria queda doblado, debido a la necesidad de acceder primero a la tabla de páginas. Una forma de evitarlo podría ser la de tener guardada la tabla de páginas en un conjunto de registros rápidos en lugar de sobre la memoria ordinaria. Sin embargo, el tamaño de la tabla de páginas es proporcional al tamaño del espacio de direcciones virtuales; de ahí que el número de registros necesarios sea demasiado grande para que esta alternativa resulte económicamente viable.

La solución al problema consiste en adoptar una técnica diferente para acceder a las páginas activas. Esta técnica representa el tener que incorporar a la máquina una *memoria asociativa*, que consistirá en un pequeño conjunto de registros de dirección de página (PARs, del inglés *page address registers*), cada uno de los cuales contiene el número de página de una página activa. Los PARs presentan la propiedad de poderse buscar en ellos de forma simultánea el número de página asociado a una dirección de programa en particular.

Así, por ejemplo, en la Figura 16, la dirección de programa 3243 se divide en el número de página 3 y el número de palabra 243 (vamos a suponer, por comodidad, que el tamaño de la página sea 1000). El número de página se compara entonces de forma simultánea con el contenido de todos los PARs, y se encuentra que coincide con el valor del PAR 5. Ello indica que la página 3 ocupa en la actualidad la página física número 5, de forma que la dirección buscada será la 5243.

El empleo de un almacenamiento de tipo asociativo reduce el tiempo empleado en la transformación de direcciones en un orden de magnitud con respecto al caso en el que se guardaba la tabla de páginas sobre memoria principal. Con el fin de que se pueda hacer referencia a todas las páginas activas a través de un PAR, hay que disponer de tantos como marcos haya en la memoria. Ello es posible en sistemas con una memoria principal reducida (como por ejemplo, el Atlas), pero en sistemas mayores no es viable, desde el punto de vista económico, disponer de todos los PARs necesarios para ello.

Se puede llegar a una solución de compromiso, guardando para cada proceso una tabla de páginas completa en la memoria, y utilizando una pequeña memoria asociativa para referenciar unas pocas páginas asociadas a los procesos activos más recientes. En este caso, el marco al que hará referencia cada PAR, no vendrá implícito por la situación de éste, sino que deberá incluirse como un campo adicional en el mismo PAR. El *hardware* de direccionamiento de la memoria lleva a cabo, entonces, la operación de transformación de direcciones que se muestra en la Figura 17. Como antes, sólo se requiere la intervención del *software* en el caso de que haya que sustituir una página.

Un problema que no ilustra la Figura 16 es el de distinguir en la memoria asociativa las páginas asociadas al proceso en ejecución de las páginas correspondientes a otros procesos. Una solución consistiría en ampliar los PARs para incluir la identificación de los procesos, junto con el número de la página. Cada dirección que se presente a la memoria asociativa deberá incluir, según esto, el identificador del proceso junto con los bits de la página.

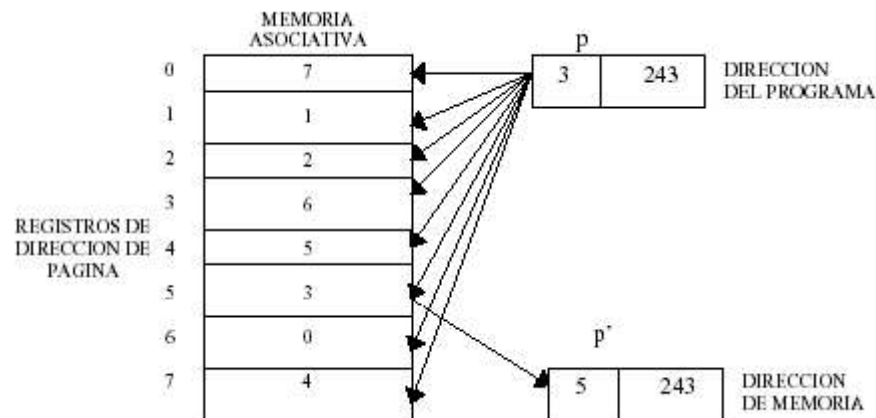


Figura 16. Transformación de direcciones mediante una memoria asociativa.

Páginas compartidas:

Otra ventaja de la paginación es la posibilidad de compartir programas de uso corriente. Esto es particularmente importante en un entorno de tiempo compartido. Consideremos un sistema que soporta 40 usuarios, cada uno de los cuales ejecuta un editor de textos. Si el editor de textos consta de 30K de código y 5K de espacio para datos, necesitaríamos 1400K para permitir a los 40 usuarios. No obstante, si el programa es *reentrante*, podría compartirse como se muestra en la Figura 18.

Aquí vemos un editor de tres páginas que es compartido por tres procesos. Cada proceso tiene su propia página de datos. El código reentrante (también llamado código puro) es un código no automodificable. Si el código es reentrante, entonces nunca cambia durante la ejecución. Así, dos o más procesos pueden ejecutar el mismo código al mismo tiempo. Cada proceso, para su ejecución, tiene su PCB y su memoria para mantener los datos. Por supuesto, los datos de todos esos procesos diferentes varían para cada uno de ellos.

Así, para permitir 40 usuarios, precisamos solamente una copia del editor, 30K, más 40 copias del espacio de 5K por usuario. El espacio total requerido es ahora de 230K, en lugar de 1400K, un ahorro significativo.

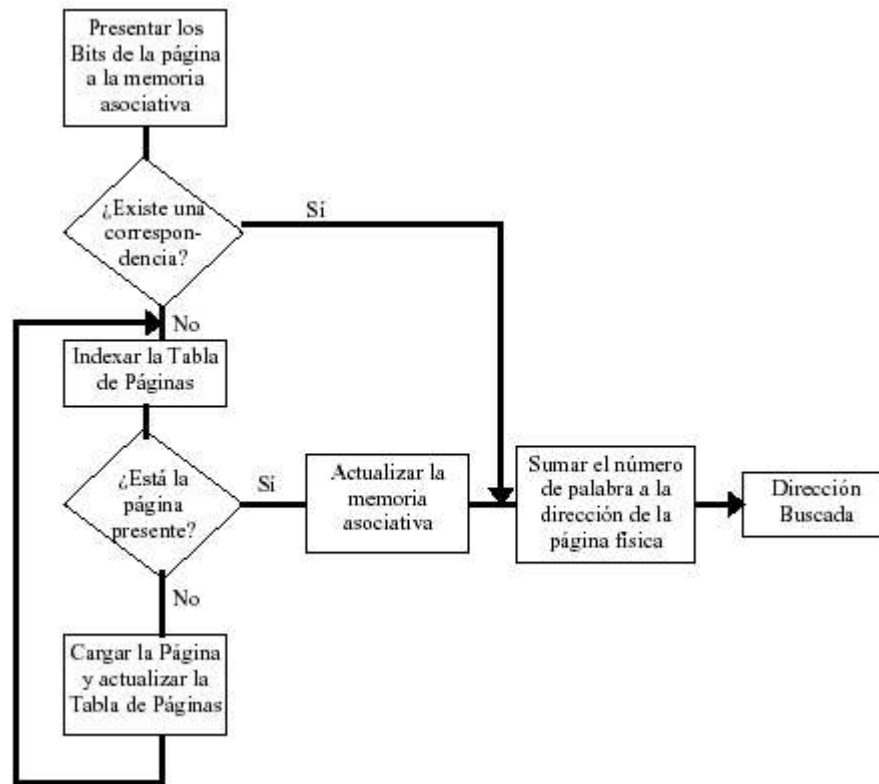


Figura 17. Transformación de direcciones para realizar la paginación con una pequeña memoria asociativa.

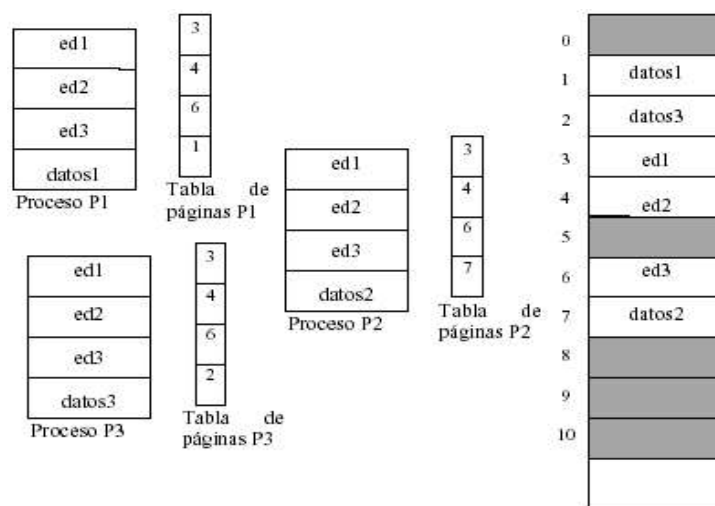


Figura 18. Compartiendo código en un entorno de paginación.

Segmentación:

La *segmentación* es un esquema de administración de la memoria que permite la visión que el usuario tiene de la misma. Un espacio de direcciones lógicas es una colección de segmentos. Cada segmento tiene un nombre y una longitud. Las direcciones especifican tanto el nombre del segmento como el desplazamiento dentro del segmento. Por lo tanto, el usuario especifica cada dirección mediante dos cantidades: un nombre de segmento y un desplazamiento. Por simplicidad de implementación, los segmentos están numerados y se referencian por un número de segmento en lugar de por un nombre. Normalmente el programa de usuario se ensambla (o compila), y el ensamblador (o el compilador) construye automáticamente segmentos que reflejan el programa de entrada.

Un compilador de Pascal podría crear segmentos separados para; las variables globales; la pila de llamada a procedimientos, para almacenar parámetros y devolver direcciones; el código de cada procedimiento o función; y las variables locales de cada procedimiento y función. El cargador tomaría todos esos segmentos y les asignaría números de segmento.

La transformación se efectúa por medio de una tabla de segmentos. El empleo de una tabla de segmentos se muestra en la Figura 19. Una dirección lógica consta de dos partes: un número de segmento s y un desplazamiento dentro de ese segmento, d . El número de segmento se utiliza como un índice en la tabla de segmentos. Cada entrada de la tabla de segmentos tiene una *base* de segmento y un *límite*. El desplazamiento d de la dirección lógica tiene que estar comprendido entre 0 y el límite de segmento. En caso contrario se produce una excepción al sistema operativo (intento de direccionamiento lógico más allá del fin del segmento). Si este desplazamiento es legal, se añade a la base para producir la dirección de la tabla deseada en la memoria física. La tabla de segmentos es así esencialmente una matriz de pares registros base/límite.

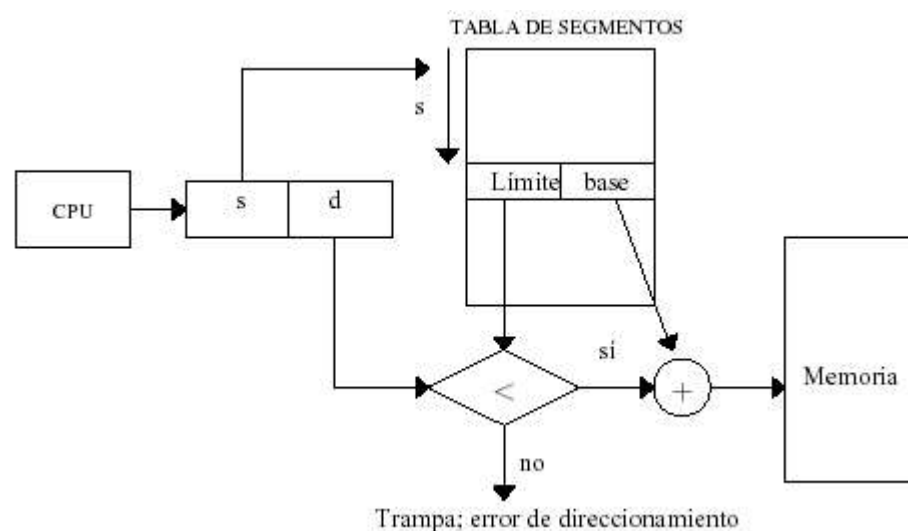


Figura 19. Hardware de segmentación.

Al igual que la tabla de páginas, la tabla de segmentos puede situarse bien en registros rápidos o bien en memoria. Una tabla de segmentos mantenida en registros puede ser referenciada muy rápidamente: la adición a la base y la comparación con el límite pueden realizarse simultáneamente para ahorrar tiempo.

Un *registro de base de tabla de segmentos* (STBR) apunta a la tabla de segmentos. Puesto que el número de segmentos utilizado por un programa puede variar ampliamente, también se utiliza un *registro de longitud de tabla de segmentos* (STLR). En el caso de una dirección lógica (s , d) verificamos primero que el número de segmento s es legal ($s < \text{STLR}$). Entonces, añadimos el número de segmento al STBR resultando la dirección en memoria de la entrada de la tabla de segmentos ($\text{STBR} + s$). Esta entrada se lee en la memoria y actuamos igual que antes: se verifica el desplazamiento frente a la longitud de segmento, y se calcula la dirección física de la palabra deseada como la suma de la base del segmento y el desplazamiento.

Igual que con la paginación, esta transformación requiere dos referencias a memoria por dirección lógica, el ordenador disminuirá su velocidad en un factor de 2, a menos que se haga algo para evitarlo. La solución normal consiste en utilizar un conjunto de registros asociativos para mantener las entradas utilizadas más recientemente en la tabla de segmentos. Un conjunto de registros asociativos relativamente pequeño (8 / 16) puede reducir generalmente el retardo a los accesos a memoria hasta no más allá de un 10% o 15% más lentos que los accesos a memoria “mapeada”.

Otra ventaja de la segmentación está relacionada con la compartición de código y datos. Los segmentos se comparten cuando las entradas en las tablas de segmentos de dos procesos diferentes apuntan a las mismas posiciones físicas. La compartición se produce a nivel de segmento. Por lo tanto, cualquier información puede compartirse definiéndole un segmento. Pueden compartirse varios segmentos, de modo que es posible compartir un programa compuesto de más de un segmento.

Fragmentación:

El sistema operativo tiene que encontrar y asignar memoria para todos los segmentos de un programa de usuario. Esta situación es similar a la paginación, excepto en el hecho de que los segmentos son de longitud *variable*, mientras que las páginas son todas del mismo tamaño. Por tanto, como en el caso de las particiones dinámicas, la asignación de memoria es un problema de asignación dinámica de almacenamiento, que se resolverá mediante un algoritmo del mejor o primer ajuste.

La segmentación puede ocasionar entonces fragmentación externa, cuando todos los bloques libres de memoria son demasiado pequeños para acomodar a un segmento. En este caso, el proceso puede simplemente verse obligado a esperar hasta que haya disponible más memoria (o al menos huecos más grandes), o puede utilizarse la compactación para crear huecos mayores. Puesto que la segmentación es por naturaleza un algoritmo de reubicación dinámica, podemos compactar la memoria siempre que queramos.

La fragmentación externa en un esquema de segmentación es mala dependiendo principalmente del tamaño

medio de segmento. En un extremo, se podría definir cada proceso como un segmento; este esquema es el de las particiones dinámicas. En el otro extremo, cada palabra podría situarse en su propio segmento y reubicarse por separado. Esta disposición elimina la fragmentación externa. Si el tamaño medio de segmento es pequeño, la fragmentación externa también será pequeña. Puesto que los segmentos individuales son más pequeños que el proceso en conjunto, es más probable que encajen en los bloques de memoria disponibles.

Segmentación paginada:

Tanto la paginación como la segmentación tienen sus ventajas y desventajas. También es posible combinar estos dos esquemas para mejorar ambos. La solución adoptada fue paginar los segmentos. La paginación elimina la fragmentación externa y convierte en trivial el problema de la asignación: cualquier marco vacío puede utilizarse para una página. Obsérvese que la diferencia entre esta solución y la segmentación pura es que la entrada en la tabla de segmentos no contiene la dirección de la base del segmento, sino la dirección de la base de una tabla de páginas para ese segmento. El desplazamiento del segmento se fragmenta entonces en un número de página de 6 bits y un desplazamiento de página de 10 bits. El número de página indexa en la tabla de páginas para dar el número de marco. Finalmente, el número de marco se combina con el desplazamiento de página para formar la dirección física.

Ahora debemos tener una tabla de páginas independiente para cada segmento. No obstante, puesto que cada segmento tiene una longitud limitada por su entrada en la tabla de segmentos, la tabla de páginas no tiene por qué tener su tamaño máximo. Sólo precisa tantas entradas como se necesiten realmente. Además, generalmente la última página de cada segmento no estará totalmente llena. De este modo tendremos, por término medio, media página de fragmentación interna por segmento. Consecuentemente, aunque hemos eliminado la fragmentación externa, hemos introducido fragmentación interna e incrementado la sobrecarga de espacio de la tabla.



1.3.7 Gestión De Discos.

Actualmente, los discos son, por lo menos, cuatro veces más lentos que la memoria principal. Este avance se espera que continúe en el futuro inmediato. De este modo, el rendimiento de los subsistemas de almacenamiento de disco es de una importancia vital, y se han realizado muchas investigaciones sobre la manera de mejorar dicho rendimiento.

Parámetros de disco.

Los detalles reales de las operaciones de E/S con los discos dependen del computador, el sistema operativo, y la naturaleza del canal de E/S y el hardware controlador de disco. La Figura 20 es una representación esquemática de la vista lateral de un *disco de cabeza móvil*. Los datos se graban sobre una serie de discos magnéticos o *platos*. Estos discos están conectados por un eje común que gira a una velocidad muy alta (por ejemplo a 3600 revoluciones por segundo).

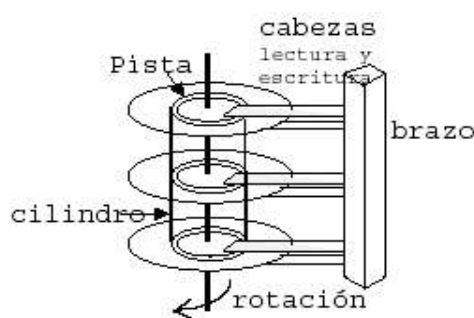


Figura 20. Mecanismo de un disco de cabeza móvil.

Cuando la unidad de disco está operando, el disco gira a una velocidad constante. Para leer o escribir, la cabeza debe posicionarse en la pista deseada, al comienzo del sector pertinente. Si el sistema es de cabezas móviles, hay que mover la cabeza para elegir la pista. Si el sistema es de cabezas fijas, habrá que seleccionar electrónicamente una de ellas. En un sistema de cabezas móviles, el tiempo que se tarda en ubicar la cabeza en la pista se llama **tiempo de búsqueda**. En cualquier caso, una vez que se ha seleccionado la pista, el controlador de disco esperará hasta que el sector apropiado se alinee con la cabeza en su rotación. El tiempo que tarda el comienzo del sector en llegar hasta la cabeza se conoce como **tiempo de latencia**. La suma del tiempo de búsqueda y de latencia es el tiempo de acceso, es decir, el tiempo que se tarda en llegar a la posición de lectura o escritura. Una vez que la cabeza está ubicada, se puede llevar a cabo la operación de Lectura o Escritura a medida que el sector se mueve bajo la cabeza; está es la parte de transferencia real de datos de la operación (**tiempo de transferencia**).

Además del tiempo de acceso y del tiempo de transferencia, en una operación de E/S intervienen algunos retardos. Cuando un proceso emite una petición de E/S, primero debe esperar en una cola a que el dispositivo esté disponible. En ese momento, el dispositivo queda asignado al proceso. Si el dispositivo comparte un único canal de E/S o un conjunto de canales con otras unidades de disco, puede producirse una espera adicional hasta que el canal esté disponible. En ese punto se realizará la búsqueda con que comienza el acceso al disco. En cualquier caso, podríamos decir que, en general, para obtener acceso a un registro de datos en particular del disco son necesarias varias operaciones. Primero, el brazo móvil debe desplazarse hacia el cilindro (o pista) apropiado (tiempo de búsqueda). Después, la porción de disco en donde se encuentran los datos debe girar hasta quedar inmediatamente abajo (o arriba) de la cabeza de lectura-escritura (tiempo de latencia). Después, el registro debe girar para pasar por la cabeza para ser leído o escrito (tiempo de transferencia). Como cada una de estas operaciones implica movimiento mecánico, el tiempo total de acceso a un registro es muy superior al tiempo de procesamiento.

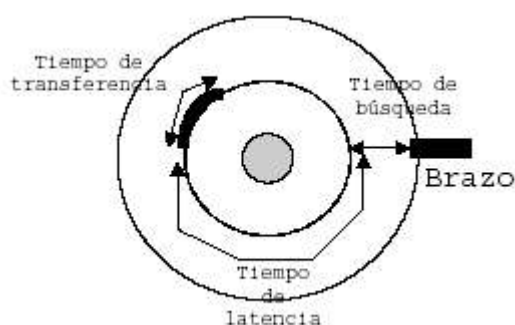


Figura 21. Componentes de un acceso a disco.

Tiempo de Búsqueda.

El tiempo de búsqueda es el tiempo necesario para mover el brazo del disco hasta la pista (o cilindro) solicitada. Esta cantidad resulta difícil de concretar. El tiempo de búsqueda consta de dos componentes clave: el tiempo de arranque inicial y el tiempo que se tarda en recorrer los cilindros, una vez que el brazo haya cogido velocidad. Por desgracia, el tiempo de recorrido no es una función lineal del número de pistas. Se puede aproximar el tiempo de búsqueda con la fórmula lineal:

$$T_b = m \times n + s$$

donde:

T_b = tiempo de búsqueda estimado n = número de pistas recorridas

m = constante que depende de la unidad de disco s = tiempo de arranque

Por ejemplo, un disco Winchester económico en un computador personal podría tener, aproximadamente, $m=0'3$ ms y $s=20$ ms, mientras que uno más grande y más caro podría tener $m=0'1$ ms y $s=3$ ms.

Tiempo de Latencia.

Los discos, excepto los flexibles, giran normalmente a 3600 rpm, es decir, una revolución cada $16'7$ ms. Por tanto, el retardo medio de giro será de $8'3$ ms. Los discos flexibles giran mucho más lentamente, generalmente entre 300 y 600 rpm. Por tanto, el retardo medio estará entre 100 y 200 ms.

Tiempo de Transferencia.

El tiempo de transferencia con el disco depende de la velocidad de rotación de la forma siguiente:

$$T_t = b / (rN)$$

donde:

T_t = tiempo de transferencia B = número de bytes a transferir

N = número de bytes por pista r = velocidad de rotación en revoluciones por segundo

Por tanto, el tiempo medio de acceso total puede expresarse como:

$$T_{at} = T_b + T_l + T_t = (m \times n + s) + (1/2r) + (b / rN)$$

donde T_b es el tiempo de búsqueda, T_l el tiempo de latencia y T_t el de transferencia.

Comparativa de Tiempos.

Considérese un disco típico con un tiempo medio de búsqueda conocido de 20 ms, velocidad de transferencia de 1 MB/sg y sectores de 512 bytes, habiendo 32 sectores por pista. Supóngase que se desea leer un fichero que consta de 256 sectores para formar un total de 128 KB. Así podría estimarse el tiempo total que dura la transferencia.

En primer lugar, supóngase que el fichero se almacena en disco de la forma más compacta posible. Es decir, el fichero ocupará todos los sectores de ocho pistas adyacentes ($8 \text{ pistas} \times 32 \text{ sectores/pista} = 256 \text{ sectores}$). Esta disposición se conoce como organización secuencial. En tal caso, el tiempo que dura la lectura de la primera pista el siguiente:

$$\text{Búsqueda media } 20'0 \text{ ms} + \text{Latencia media } 8'3 \text{ ms} + \text{Lectura de 32 sectores } 16'7 \text{ ms} = 45'0 \text{ ms}$$

Supóngase que las pistas restantes pueden leerse ahora sin tiempo de búsqueda alguno. Es decir, la operación de E/S puede llevar el ritmo del flujo de datos que lleva el disco. En tal caso hace falta considerar, como mucho, el tiempo de latencia. Por tanto, cada pista consecutiva se puede leer en $8'3 + 16'7 = 25$ ms. Para leer el fichero por completo:

$$\text{Tiempo total} = 45 + 7 \times 25 = 220 \text{ ms} = 0'22 \text{ seg.}$$

Se calcula ahora el tiempo necesario para leer los mismos datos utilizando acceso aleatorio en vez de acceso secuencial; esto es, el acceso a los sectores se distribuye aleatoriamente por el disco. Para cada sector, se tiene:

$$\text{Búsqueda media } 20'0 \text{ ms} + \text{Latencia media } 8'3 \text{ ms} + \text{Lectura de 1 sector } 0'5 \text{ ms} = 28'8 \text{ ms}$$

$$\text{Tiempo total} = 256 \times 28'8 = 7373 \text{ ms} = 7'37 \text{ seg.}$$

Está claro que el orden en que se leen los sectores del disco tiene un efecto inmenso en el rendimiento de la E/S. En el caso de los accesos a ficheros en los que se leen varios sectores, se puede ejercer algún control sobre la manera en que se distribuyen los sectores de datos tal y como veremos en el tema dedicado a gestión de sistemas de ficheros. Sin embargo, incluso en el caso de acceso a un fichero en un entorno de multiprogramación, existirán varias solicitudes de E/S que compitan por el mismo disco. Por tanto, merece la pena investigar alguna manera más de mejorar el rendimiento de E/S a disco.

Políticas de planificación.

Antes de pasar a describir las distintas políticas de planificación que se pueden adoptar, vamos a enumerar los criterios que nos van a permitir valorarlas:

- la productividad
- el tiempo promedio de respuesta
- la varianza de los tiempos de respuesta (predecibilidad)

Está claro que una política de planificación debe tratar de lograr una productividad máxima (el mayor número posible de peticiones atendidas por unidad de tiempo). Para ello, se deberá reducir al máximo el tiempo desperdiciado en las búsquedas muy largas. Pero una política de planificación también debe tratar de reducir el tiempo promedio de respuesta (es decir, el tiempo promedio de espera más el tiempo promedio de servicio).

Los criterios señalados tratan de mejorar el rendimiento global, tal vez a expensas de las peticiones individuales. La planificación mejora a menudo el rendimiento global pero reduce el nivel de atención a ciertas peticiones. Una medida importante de este fenómeno es la varianza de los tiempos de respuesta. La varianza es una media

matemática de cuánto se desvían elementos individuales del promedio de los elementos. Como tal, utilizaremos la varianza para indicar la predecibilidad: a menor varianza mayor predecibilidad. Deseamos una política que reduzca al mínimo la varianza.

Planificación FCFS (First Come First Served).

Por supuesto, la forma más sencilla para planificar un disco es la planificación primera solicitud en llegar primera en ser servida (FCFS, First Come First Served). Este algoritmo es fácil de programar e intrínsecamente justo, pero es probable que no ofrezca el mejor tiempo de servicio (en promedio).

Considere, por ejemplo, una cola de peticiones a disco que afecta a las pistas 98, 183, 37, 122, 14, 124, 65 y 67. Si, en principio, la cabeza de lectura-escritura está en la pista 53, se moverá primero de la 53 a la 98, luego a la 183, 37, 122, 14, 124, 65 y por último a la 67, lo que da un movimiento total de la cabeza de 640 pistas (Figura 22).

FCFS es justa en el sentido de que al llegar una solicitud, su lugar en la planificación es fijo (se respeta el orden de llegada). Una petición no puede ser desplazada por la llegada de otra con mayor prioridad. Cuando las peticiones están distribuidas uniformemente en la superficie del disco, la planificación FCFS da como resultado un patrón de búsqueda aleatorio. Hace caso omiso de la relaciones de posición entre las solicitudes pendientes. No hace ningún intento por optimizar el patrón de búsqueda.

FCFS es aceptable cuando la carga de disco es ligera. Pero conforme crece la carga, FCFS tiende a saturar el dispositivo, y aumenta los tiempos de respuesta. FCFS ofrece una varianza pequeña, pero esto no es un gran consuelo para la petición que espera al final de la cola mientras el brazo móvil se desplaza de un lado a otro.

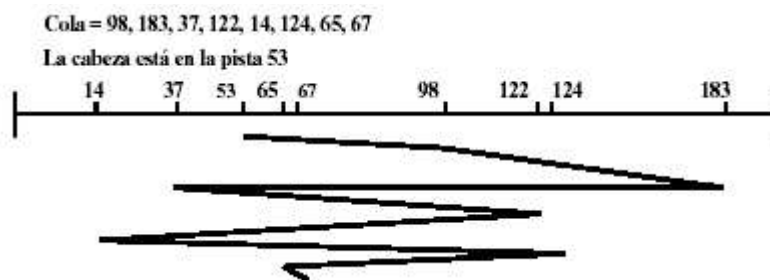


Figura 22. Planificación de disco First-Come-First-Served.

Planificación SSTF (Shortest Seek Time First).

Parece razonable servir juntas todas las solicitudes próximas a la posición actual del disco, antes de desplazar la cabeza a un punto lejano para servir otra solicitud. Esta suposición es la base del algoritmo primero la de menor tiempo de posicionamiento (SSTF, *Shortest Seek Time First*) para la planificación de disco. El algoritmo SSTF selecciona la solicitud con menor tiempo de posicionamiento a partir de la posición actual de la cabeza. Como el tiempo de búsqueda normalmente es proporcional a la diferencia entre las solicitudes medida en pistas, implantamos esta estrategia moviendo la cabeza a la pista más próxima de la cola de solicitudes.

En nuestro ejemplo de cola de solicitudes, la más próxima a la posición inicial (53) es la pista 65. Una vez ubicados en la pista 65, la siguiente pista más cercana es la 67. En la figura aparece el orden en el que se resolverían las peticiones. Este método de planificación da como resultado un movimiento total de sólo 236 pistas, poco más de la tercera parte de la distancia recorrida con la planificación FCFS. Este algoritmo mejora considerablemente el promedio del servicio del disco.

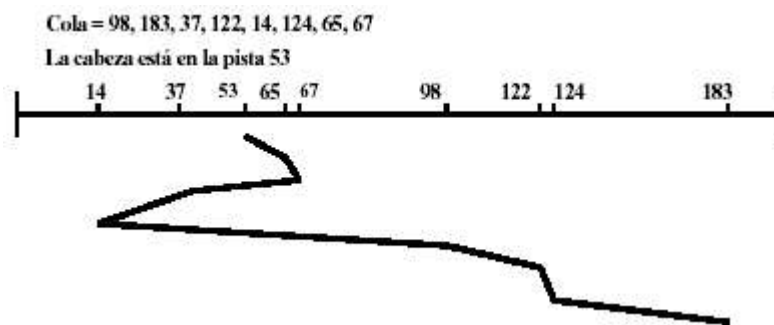


Figura 23. Planificación de disco Shortest-Seek-Time-First.

SSTF tiende a favorecer mucho ciertas peticiones. Los patrones de búsqueda SSTF tienden a estar muy localizados y, en consecuencia, las pistas más exteriores e interiores pueden recibir una atención deficiente en comparación con la que reciben las pistas de la parte media.

SSTF ofrece mejores tasas de productividad que FCFS pero no es el algoritmo óptimo. Por ejemplo, si movemos la cabeza de la 53 a la 37, aunque ésta no sea la pista más próxima, y luego a la 14, antes de regresar para servir a la 65, 67, 98, 122 y 124 y 183 podemos reducir el movimiento total de la cabeza a 208 pistas. Los tiempos de respuesta tienden a ser más bajos cuando la carga es moderada. Una desventaja importante es que aumenta la varianza de los tiempos de respuesta debido a la discriminación contra las pistas exteriores e interiores. En un caso extremo, su gestión podría sufrir un aplazamiento indefinido. Si se consideran las importantes mejoras en la productividad y en los tiempos promedio de respuesta, el aumento de la varianza puede parecer tolerable. SSTF resulta útil en sistemas de procesamiento por lotes, donde la productividad es lo más importante. Pero la elevada varianza de los tiempos de respuesta (su impredecibilidad) lo hace inaceptable en sistemas interactivos.

Planificación SCAN.

Denning (1967) desarrolló la estrategia de planificación SCAN para evitar la discriminación y la alta variación en los tiempos de respuesta de SSTF. SCAN opera como SSTF, excepto que SCAN elige la solicitud que implica la menor distancia de búsqueda en una dirección predefinida. Si la dirección predefinida es en un momento dado hacia afuera, la estrategia SCAN elige la distancia de búsqueda más corta en esa dirección. SCAN no cambia de dirección hasta que llega a la pista más exterior o hasta que ya no hay solicitudes pendientes en la dirección predefinida.

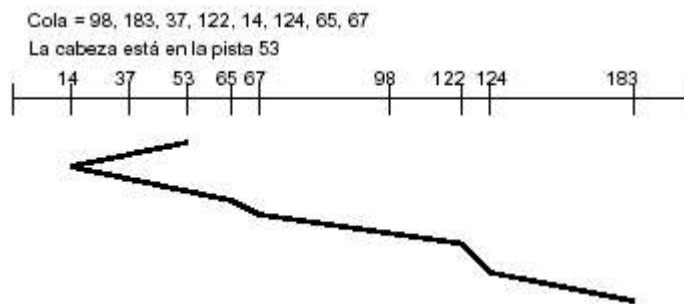


Figura 24. Planificación de disco SCAN.

Para el ejemplo que nos ocupa necesitamos saber la dirección de movimiento de la cabeza, así como su posición más reciente. Aquí hemos supuesto que la cabeza se desplaza hacia las pistas de menor número, y se parte de la 53. SCAN tiene un comportamiento muy parecido al de SSTF en términos de aumento de productividad y reducción de tiempos promedio de espera, pero elimina gran parte de la discriminación inherente a los esquemas SSTF y ofrece una varianza mucho menor. A causa del movimiento oscilante de las cabezas de lectura-escritura en SCAN, las pistas más exteriores se visitan con menos frecuencia que las de la parte media, pero ello no es tan grave como la discriminación de SSTF.

Planificación N-SCAN.

Una modificación interesante de la estrategia SCAN básica se denomina SCAN de N pasos. En esta estrategia, el brazo del disco se mueve en una y otra dirección como en SCAN, excepto que sólo atiende las solicitudes que ya estaban esperando cuando se inició un barrido específico. Las solicitudes que llegan durante una barrido se agrupan para darles una atención óptima durante el barrido de regreso. En cada barrido se atienden las primeras N peticiones.

SCAN de N pasos ofrece un buen desempeño en cuanto a productividad y tiempo promedio de respuesta. Su característica más importante es una menor variación de los tiempos de respuesta que en la planificación SSTF o en la SCAN convencional. SCAN de N pasos elimina la posibilidad de que ocurra un aplazamiento indefinido si llega un gran número de peticiones para la pista actual. Éste guarda dichas peticiones para atenderlas en el siguiente barrido.

Planificación C-SCAN.

Otra modificación interesante de la estrategia SCAN básica se denomina CSCAN (SCAN circular). C-SCAN elimina la discriminación de las estrategias anteriores contra las pistas más interiores y exteriores.

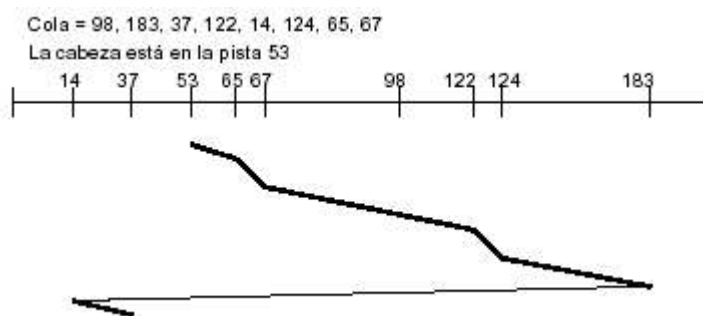


Figura 25. C-SCAN

En la estrategia C-SCAN el brazo se mueve del cilindro exterior hacia el interior, atendiendo solicitudes según un criterio de búsqueda más corta. Cuando el brazo ha completado su barrido hacia adentro, salta (sin atender peticiones) a la solicitud más cercana al cilindro más exterior y luego reinicia su barrido hacia adentro procesando solicitudes. C-SCAN se puede llevar a la práctica de manera tal que las solicitudes que lleguen durante un barrido sean atendidas en el siguiente barrido. Presenta una varianza muy pequeña con respecto a los tiempos de respuesta. Algunos resultados de simulaciones indican que la mejor política de planificación de disco podría operar en dos etapas. Cuando la carga es ligera, la política SCAN es la mejor. Cuando la carga es mediana o pesada, C-SCAN produce los mejores resultados.

Caché de disco.

El término memoria caché se aplica normalmente a una memoria más pequeña y rápida que la memoria principal, que se sitúa entre ésta y el procesador. Este tipo de memoria caché reduce el tiempo medio de acceso a memoria aprovechándose del principio de localidad. El mismo principio puede aplicarse a la memoria de disco. Más en concreto, una caché de disco es un buffer para sectores de disco situado en la memoria principal. La caché contiene una copia de algunos sectores del disco. Cuando se hace una petición de E/S para un bloque, se comprueba si ese bloque está en la caché del disco. Si es así, la petición se cumple con la caché. Si no, se lee el sector pedido de disco y se coloca en la caché. Debido a la localidad de referencias o localidad, cuando se traiga un bloque de datos a la caché para satisfacer una sola petición de E/S, será probable que se produzcan referencias futuras al mismo bloque. Como las operaciones de escritura no siempre se realizan cuando los programas creen que se realizaron, es probable que en un momento dado el contenido de los ficheros de disco difiera de lo que los programas creen que contienen. Este problema es crucial sobre todo si se cae el sistema. Para reducir al mínimo la posibilidad de que los discos estén

en desacuerdo con lo que los programas creen que contienen, el sistema operativo UNIX ejecuta periódicamente la llamada al sistema *sync* para grabar todos los buffers de disco.

Consideraciones de Diseño

Son de interés varias cuestiones de diseño. En primer lugar, cuando una petición de E/S se sirve con la caché, los datos de la misma deben ser enviados al proceso que los solicitó. El envío puede hacerse por una transferencia en memoria del bloque de datos, desde la caché del disco a la memoria asignada al proceso de usuario o, simplemente, usar la posibilidad de memoria compartida y pasar un puntero a la entrada apropiada de la caché del disco. Este último método ahorra el tiempo de transferencia interna a memoria y, además, permite el acceso compartido de otros procesos que puedan seguir el modelo de los lectores/escritores descrito en el tema de concurrencia.

Una segunda cuestión de diseño tiene que ver con la estrategia de reemplazo. Cuando se trae un nuevo bloque a la caché del disco, uno de los bloques existentes debe ser sustituido. Este problema es idéntico al presentado en el tema de memoria virtual, donde se empleaban algoritmos de reemplazo de páginas. Se han probado un buen número de algoritmos. El algoritmo más común es el LRU, en el que se reemplaza el bloque que lleva más tiempo en la caché sin ser referenciado. Sin considerar la estrategia de reemplazo en particular, la sustitución puede llevarse a cabo bajo demanda o puede ser planificada previamente. En el primer caso, los sectores se sustituyen sólo cuando se necesita su entrada de la tabla. En el último caso, cada vez se liberan un conjunto de entradas. La razón de ser de este método está relacionada con la necesidad de volver a escribir los bloques. Si se trae un bloque a la caché y sólo es leído, no será necesario volver a escribirlo al disco cuando sea reemplazado. Sin embargo, si el bloque es actualizado, entonces sí será necesario volver a escribirlo antes de reemplazarlo. En este último caso, tiene sentido agrupar las escrituras y ordenarlas para minimizar el tiempo de búsqueda.

Consideraciones de Rendimiento

Aquí se pueden aplicar las mismas consideraciones sobre el rendimiento discutidas en el tema de administración de memoria virtual. El rendimiento de la caché será bueno siempre que sea mínima la tasa de fallos. Esto dependerá de la localidad de las referencias al disco, del algoritmo de reemplazo y de otros factores de diseño. Sin embargo, la tasa de fallos es principalmente función del tamaño de la caché de disco.



1.3.8 Entrada / Salida.

La gestión de E/S está basada en la consecución de varios objetivos, como; la independencia del código de los caracteres, independencia del periférico, eficiencia y tratamiento uniforme de los periféricos.

Algunas de las consecuencias de estos objetivos son evidentes. En primer lugar, la independencia del código de los caracteres implica la existencia de una representación interna uniforme de todos ellos. Esta representación se denomina el *código interno de los caracteres*, debiéndose asociar a cada periférico los mecanismos adecuados de traducción con el fin de que se lleven a cabo las conversiones adecuadas a las entradas y salidas. Para los periféricos que soportan varios códigos de caracteres hay que disponer de distintos mecanismos de traducción para cubrir los distintos códigos soportados. Esta traducción se efectuará inmediatamente después de la entrada e inmediatamente antes de la salida, de forma que los procesos íntimamente ligados a la gestión de los periféricos deberán conocer el código intermedio empleado. El mecanismo de traducción estará basado generalmente en una tabla, o en algunos casos, en un pequeño fragmento de programa.

Además, la independencia del periférico implica que los programas deberían trabajar no sobre periféricos físicos, sino sobre periféricos virtuales, denominados unas veces *streams* y otras ficheros, o bien conjuntos de datos. Estos *streams* se emplean en un programa sin que se haga referencia alguna a los periféricos físicos: el programador simplemente dirige sus entradas o salidas hacia o desde uno de estos *streams* en particular.

La correspondencia entre *streams* y tipos de periféricos para un proceso en particular puede guardarse en una lista de *descriptores de streams*, a la que apunta el descriptor de proceso correspondiente (Figura 26). La asignación de un modelo particular de un determinado tipo de periférico se realiza cuando el proceso utiliza por primera vez el correspondiente *stream*. Diremos que en este momento el proceso *abre* el *stream*. Este *stream* se *cierra*, lo que significa que ya no puede ser utilizado más, por acción explícita del proceso, o bien de forma implícita al finalizar éste.

La correspondencia entre *streams* y tipos de periféricos para un proceso en particular puede guardarse en una lista de *descriptores de streams*, a la que apunta el descriptor de proceso correspondiente (Figura 26). La asignación de un modelo particular de un determinado tipo de periférico se realiza cuando el proceso utiliza por primera vez el correspondiente *stream*. Diremos que en este momento el proceso *abre* el *stream*. Este *stream* se *cierra*, lo que significa que ya no puede ser utilizado más, por acción explícita del proceso, o bien de forma implícita al finalizar éste.



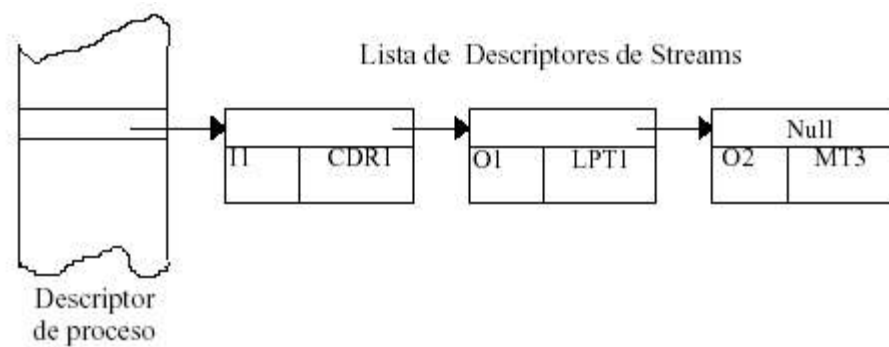


Figura 26. Información acerca de los streams y los periféricos asociados a un proceso.

Una tercera consecuencia de los objetivos de diseño que citamos es la de que el sistema de E/S debería construirse de forma que las características de los periféricos estuviesen claramente asociadas a los propios periféricos en lugar de estarlo a las rutinas que los gestionan (los *gestores de periféricos*). De esta forma sería posible que estos gestores de periféricos mostrasen grandes semejanzas entre ellos y que sus diferentes formas de operación derivasen tan sólo de información de tipo paramétrico, que se pudiese obtener de las características de cada periférico en particular. El necesario aislamiento de estas características de los periféricos puede obtenerse a través de su codificación en sendos *descriptores de periférico* que se asocien a cada uno de ellos. Estos descriptores constituirán la fuente de información de los diferentes programas de gestión de los periféricos. La información sobre un periférico que puede almacenarse en su descriptor es:

1. la identificación del periférico.
2. las instrucciones a través de las que actúa.
3. punteros a tablas de traducción de caracteres.
4. el *status* actual: si está ocupado, libre o estropeado.
5. el proceso de usuario en curso: un puntero al descriptor de proceso que esté utilizando el periférico en cuestión.

Las rutinas de E/S

Estamos ahora en condiciones de ver cómo gestiona el sistema operativo una petición de E/S procedente de un proceso de usuario.

Una petición típica de uno de estos procesos consistirá en una llamada al sistema operativo con la estructura general siguiente:

DOIO (*stream*, *modo*, *cantidad*, *destino*, *semáforo*) donde:

- DOIO es el nombre de una rutina de E/S del sistema.
- *stream* es el número del *stream* en el que debe tener lugar la E/S.
- *modo* indica qué operación hay que realizar (tal como transferir datos). También puede indicar, si ello es relevante, el código de caracteres a emplear.
- *cantidad* es la cantidad de información a transferir .
- *destino* (o fuente) es la posición en la que (o desde que) debe efectuarse la transferencia.
- *semáforo* es la dirección del semáforo "petición servida" que deberá activarse cuando se complete la operación de E/S.

La rutina de E/S DOIO es reentrante, de forma que puede ser empleada por varios procesos a la vez. Su función es la de asignar el número de *stream* al periférico físico adecuado para comprobar la consistencia de los parámetros que se le suministren, así como para iniciar el servicio de la petición. La primera de estas operaciones es inmediata. El periférico que corresponde al *stream* en cuestión puede determinarse a partir de la información almacenada en la lista del descriptor de proceso de llamada (ver la Figura 26). Una vez identificado el periférico pueden comprobarse los parámetros de la petición de E/S de acuerdo con la información guardada en el descriptor de periférico, y si se detecta un error se puede volver al programa de llamada. Una comprobación en concreto que puede llevarse a cabo es la de que sea capaz el periférico de trabajar en el modo deseado; otra es la de que el tamaño y destino de la transferencia de datos se correspondan con el modo de operación seleccionado. En el caso de aquellos periféricos que pueden transferir sólo caracteres aislados, la cantidad a especificar deberá ser igual al tamaño del bloque (fijo o variable, dependiendo del periférico) y el destino la posición de memoria a partir de la cual debe empezar a realizarse la transferencia.

Una vez efectuadas las distintas comprobaciones, la rutina de E/S agrupa los parámetros de la petición en un bloque de petición de E/S (IORB, del inglés *I/O request block*) que incorpora a una cola de bloques similares que representan otras peticiones de uso del mismo periférico. Estas otras peticiones pueden provenir del mismo proceso o, en el caso de un periférico compartido tal como un disco, pueden proceder de otros procesos. Esta cola de petición de periférico está ligada al descriptor del periférico en cuestión (Figura 27) y es atendida por un proceso independiente denominado gestor de periférico que veremos más adelante. La rutina de E/S notifica al gestor de periférico que ha colocado una petición en la cola a través de un *signal* a un semáforo de *petición pendiente* asociado al periférico en cuestión, y que está contenido en el descriptor de este periférico. De manera similar, una vez finalizada la operación

de E/S, el gestor de periférico notifica esta circunstancia al proceso del usuario a través de un semáforo de *petición servida*, semáforo cuya dirección constituía uno de los parámetros de la rutina de E/S, y había sido transferida al gestor del periférico como un elemento más del IORB. Este semáforo puede ser inicializado por el proceso del usuario o bien por la rutina de E/S al crearse el IORB.

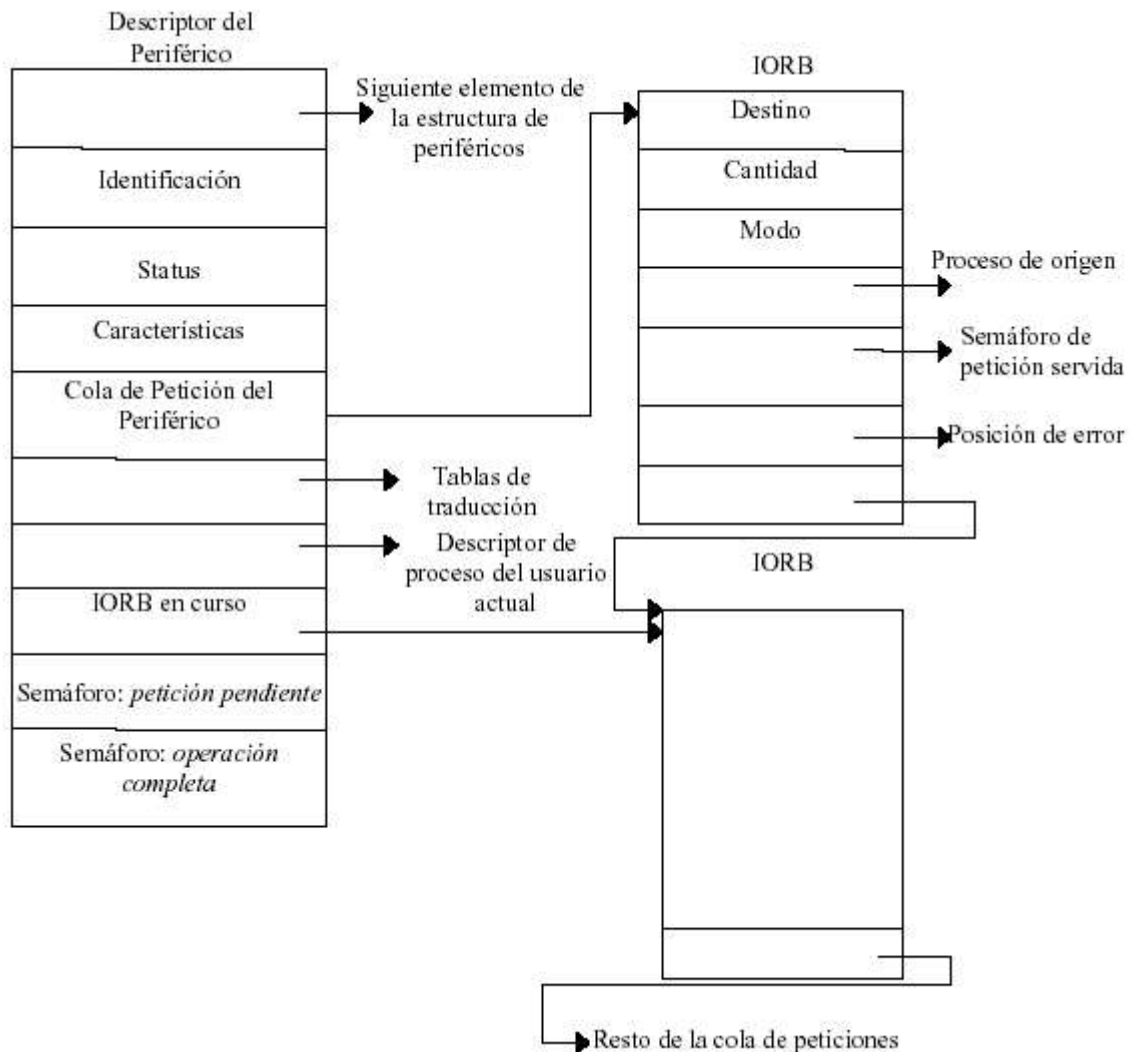


Figura 27. Descriptor y cola de petición de periférico.

La rutina completa de E/S es:

rutina DOIO (*stream, modo, cantidad, destino, semáforo*)

inicio

analizar el periférico en el descriptor de proceso;

comprobar valor de parámetros frente a características del periférico;

si error **entonces** salida de error;

construir el IORB;

incorporar el IORB a la cola de petición del periférico;

signal (petición pendiente);

final;

El control de los periféricos

Tal como indicábamos antes, un gestor de periférico es un proceso sobre el que recae la responsabilidad del servicio de las peticiones de la cola de peticiones de periférico, así como la de informar al proceso de origen del final de la transferencia. Existe un programa de gestión para cada periférico pero, como que todos ellos actúan de forma similar, pueden utilizar programas de uso compartido. Cualquier diferencia de comportamiento entre diversos gestores de periféricos deriva de la naturaleza de las características almacenadas en los descriptores de los periféricos correspondientes.

Un programa de gestión de periférico funciona según un ciclo continuo durante el cual saca un IORB de la cola de peticiones, inicia la correspondiente operación de E/S, espera a que finalice esta operación y notifica el fin de la transferencia al proceso de origen. El ciclo completo para una operación de entrada es:

repetir de forma indefinida

inicio

wait (petición pendiente);

sacar un IORB de la cola de peticiones;

obtener los detalles de la petición escogida;

iniciar la operación de E/S;



wait (operación completa)

si error entonces establecer la información asociada al error;

traducir los caracteres (si es necesario);

transferir los datos al lugar de destino;

signal (petición servida);

borrar el IORB;

final;

La sincronización y el flujo de control entre el proceso que solicita la E/S, la rutina de E/S, el gestor de periférico adecuado y la rutina de interrupción se ilustran en la Figura 28. Las flechas continuas representan transferencias de control y las flechas a trazos representan sincronizaciones por medio de semáforos. Es importante hacer notar que en este esquema el proceso que solicita la E/S se ejecuta asincrónicamente con el gestor de periférico, de forma que puede ejecutar otras operaciones o bien realizar otras peticiones de E/S mientras se esté sirviendo la petición original. El proceso que solicita la E/S deberá bloquearse sólo cuando no haya finalizado la operación de E/S al ejecutar *wait (petición servida)*.

El inconveniente de este enfoque es que el proceso que solicita la E/S debe asumir la responsabilidad de sincronizar sus actividades con las del gestor de periférico. No debe, por ejemplo, intentar utilizar información que no haya sido aún transferida. Ello significa que el autor del proceso debe ser consciente de que las operaciones de E/S no son instantáneas y debe ser capaz, pues, de lograr la sincronización deseada mediante el empleo adecuado del semáforo de *petición servida*.

Una posible alternativa consistiría en asignar la responsabilidad de la sincronización a la rutina de E/S que forma parte del sistema operativo. Ello puede llevarse a cabo haciendo que el semáforo de *petición servida* sea local a la rutina de E/S (o sea, su dirección ya no será preciso que la suministre el proceso que solicita la E/S) y añadiendo las operaciones:

wait (petición servida);

comprobar la posición de error;

En la rutina de E/S, justo antes del final. El retraso que lleva implícito una operación de E/S estará oculto ahora por el sistema operativo: por lo que concierne al proceso que solicita esta E/S, la operación será instantánea en el sentido de que cuando se ejecute la instrucción siguiente a dicha petición, la transferencia podrá suponerse

finalizada. En otras palabras, el proceso que solicita la E/S se ejecuta sobre una máquina virtual en la que estas operaciones de transferencia son realizadas por una sola instrucción de tipo instantáneo.

Las dos posibles opciones presentadas en líneas anteriores pueden resumirse como sigue. En la primera el usuario es libre de seguir ejecutando su programa en paralelo con la operación de E/S, pero es él mismo quien debe detectar el final de la misma. En el segundo caso se le libera de esta responsabilidad, aunque no se le permite ya ejecutar su programa en paralelo con la operación de transferencia.

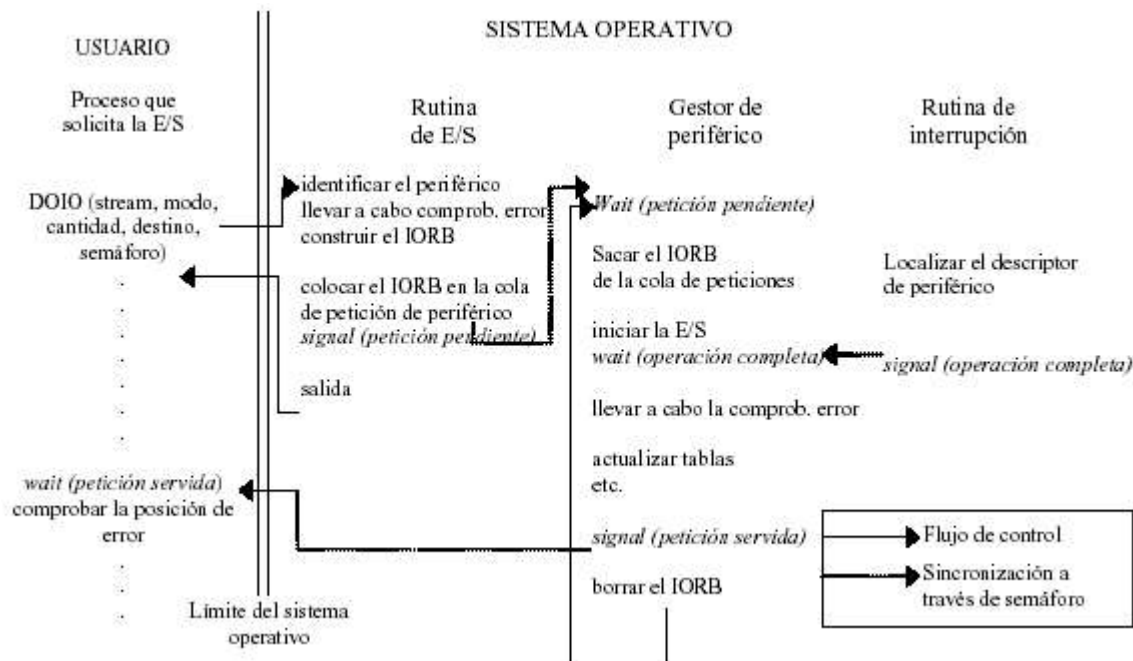


Figura 28. Esquema del sistema de E/S.

La técnica del "buffering"

En la descripción de la rutina de E/S y de los programas de gestión de los periféricos que se ha realizado en líneas anteriores se ha supuesto que todas las transferencias de datos eran directas. Esto es, cada petición de E/S por parte de un proceso provocaba que tuviese lugar una transferencia física en el periférico adecuado. Si el proceso debía realizar varias transferencias sobre el mismo *stream* se vería bloqueado repetidas veces (en *wait (petición servida)*) mientras tenía lugar la transferencia. Con el fin de evitar pérdidas importantes de tiempo en el proceso, en algunos casos es conveniente llevar a cabo las transferencias de E/S antes de que se hagan las peticiones, asegurando así que la información esté disponible cuando sea necesaria. Esta técnica recibe el nombre de *buffering*.

La idea consiste en que el sistema operativo realice las transferencias de entrada a una zona de memoria denominada *buffer* de entrada, y que el proceso del usuario coja los datos de allí, debiendo esperar sólo cuando este *buffer* esté vacío. Cuando esto ocurra, el sistema operativo volverá a llenar este *buffer* y el proceso seguirá. De forma

similar, las transferencias de salida de un proceso se dirigirán a un *buffer* de salida, siendo el sistema operativo el encargado de vaciarlo cuando esté lleno. El proceso sólo deberá esperar en el caso de que intente meter un elemento antes de que el sistema haya vaciado el *buffer*.

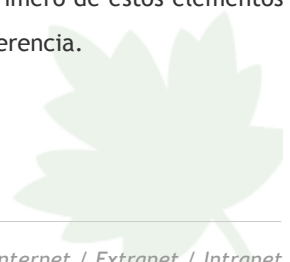
Puede depurarse esta técnica mediante el empleo de dos *buffers* en lugar de uno solo: es lo que se denomina *buffering* doble. Un proceso transferirá ahora datos a (o desde) un *buffer* mientras el sistema operativo vacía (o llena) el otro. Con ello se asegura que el proceso debe detenerse sólo si ambos *buffers* están llenos (o vacíos) antes de que el sistema operativo haya llevado a cabo su acción. Ello ocurrirá sólo cuando el proceso realice una gran cantidad de E/S y, en este caso, muy a menudo puede resolverse el problema añadiendo más *buffers* (*buffering* múltiple). Evidentemente de nada servirá todo esto en los casos en que un proceso realice sus E/S a una velocidad superior a la velocidad a la que pueden trabajar los periféricos de E/S. La utilidad de la técnica del *buffering* consiste en eliminar los efectos de picos elevados de demandas de E/S en el caso en que la demanda promedio no sea mayor que la que pueden atender los periféricos de E/S. En general, cuanto mayores sean estos picos, mayor será el número de *buffers* necesarios para conseguir el efecto deseado.

El sistema operativo asigna espacio a los *buffers* cuando se abre el *stream*, guardando en el descriptor del mismo sus respectivas direcciones.

Para permitir una transferencia a través de *buffers* es necesaria una rutina de E/S algo diferente. La nueva rutina realizará una entrada cogiendo el elemento siguiente del *buffer* apropiado y transfiriéndolo directamente al proceso de llamada. Sólo cuando el *buffer* quede vacío generará la rutina un IORB y avisará al gestor de periférico adecuado para que le suministre más información. Cuando el *stream* esté abierto, la rutina de E/S generará suficientes IORBs para llenar todos los *buffers*. El gestor de periférico trabaja como antes, iniciando una transferencia de datos al *buffer* cuya dirección viene indicada en el IORB. Lo mismo es válido *mutatis mutandis* para las transferencias de salida. En ambos casos la rutina de E/S y el gestor del periférico constituyen, juntos, una variante de la pareja productor-consumidor. La rutina de E/S para transferencias a través de *buffers* puede llamarse desde un proceso de usuario mediante una instrucción del tipo:

DOBUFFIO (*stream*, *modo*, *destino*)

donde *stream*, *modo* y *destino* tienen el mismo significado que para la rutina DOIO que se comentó anteriormente. La cantidad de información transmitida será de un sólo elemento. Es de destacar que, como cualquier retardo que aparezca como consecuencia de *buffers* llenos o vacíos queda oculto por la rutina de E/S, no hay necesidad alguna de pasar una dirección de un semáforo como uno de los parámetros. El tipo de *buffering* a emplear, si se da el caso, y las direcciones de los *buffers* y los semáforos a utilizar por las rutinas de E/S estarán todos ellos accesibles en la lista del descriptor del *stream* del proceso de llamada (Figura 29-(b)). El primero de estos elementos puede utilizarse para determinar qué rutina de E/S habrá que llamar para efectuar la transferencia.



Ficheros y periféricos

Hasta ahora se ha supuesto de forma implícita que el nombre de un periférico constituía información suficiente para determinar el origen o destino externos de una determinada transferencia. Ello es cierto para periféricos que trabajan de forma secuencial de modo que no haya ambigüedad alguna acerca de la zona del medio externo hacia o desde donde se dirige una transferencia de datos. Un teclado, por ejemplo, sólo puede leer el siguiente carácter pulsado, al igual que una impresora sólo puede escribir en la línea en curso. Otros periféricos, tal como los transportes de disco o de tambor, trabajan en base a un modo de acceso aleatorio y permiten seleccionar una zona cualquiera del medio empleado (sea un disco o tambor) para efectuar allí la transferencia. En estos casos no es suficiente dar el nombre del periférico empleado, sino que hay que especificar también la zona que hay que utilizar del medio.

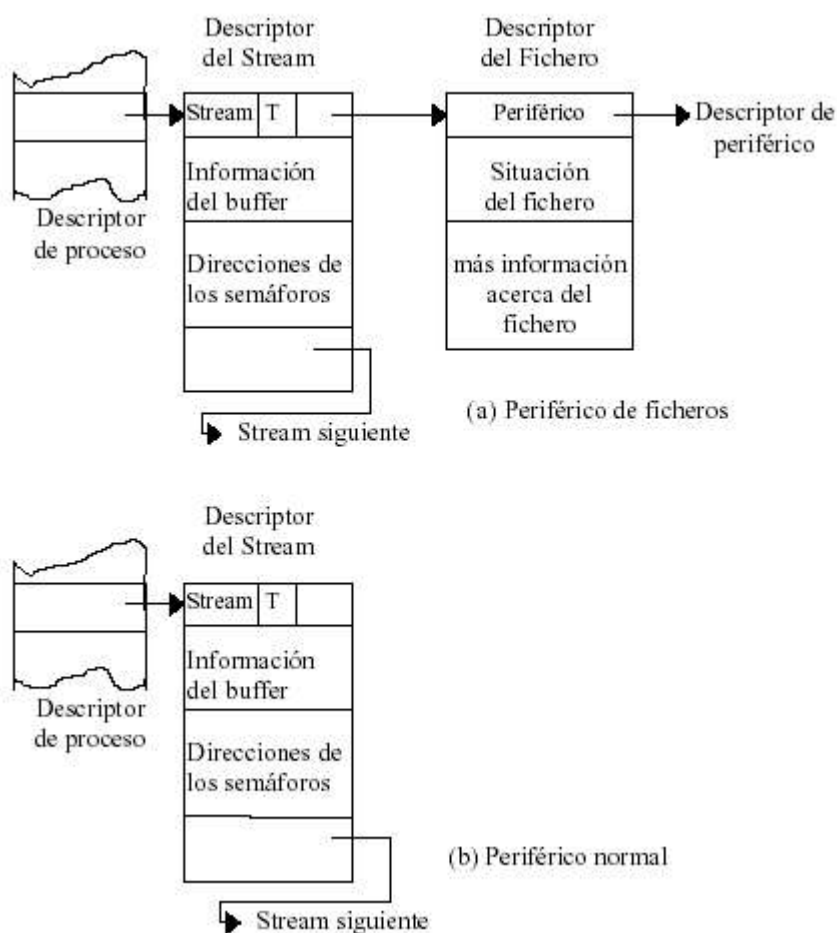


Figura 29. Información asociada a un stream en el caso de periféricos de: fichero y normal (T= tipo de periférico).

Cada zona de datos que puede existir en soportes de este tipo se denomina un *fichero*. Todo fichero posee una trayectoria única que puede utilizar el sistema operativo para determinar la situación del fichero en el medio correspondiente.

Cada vez que se abre un fichero se crea un *descriptor de fichero* que contendrá toda la información que sea necesaria para futuras transferencias. Esta información incluye la dirección del descriptor del periférico en el que está almacenado el fichero, la situación del fichero en dicho periférico, si hay que leer o escribir en él, así como otros detalles acerca de la organización interna del fichero en cuestión. Tal como se ilustra en la Figura 29-(a), se coloca un puntero que señala al descriptor del fichero en el descriptor del *stream* adecuado del proceso que abre el fichero.

Comparando la Figura 29-(a) y la Figura 29-(b) el lector podrá ver que el descriptor de fichero contiene información adicional acerca de la asociación de los *streams* con los periféricos físicos. Las rutinas de E/S que llevan a cabo esta asociación para cada transferencia de datos pueden modificarse fácilmente con el fin de que tomen esta información en cuenta al construir el IORB.

La técnica del "*spooling*"

Anteriormente hemos hecho una distinción implícita entre periféricos compartibles, como los transportes de disco, que pueden atender sucesivas peticiones procedentes de diferentes procesos, y los periféricos no compartibles, tal como las impresoras, que pueden asignarse a un sólo proceso a la vez. Estos periféricos no compartibles son aquellos que trabajan de forma tal que su asignación a diversos procesos a la vez conduciría a una mezcla caótica de operaciones de E/S. La asignación de un periférico no compartible se realiza cuando un proceso abre un *stream* que esté asociado a él. El periférico es liberado sólo cuando se cierra el *stream* o bien finaliza el proceso. Los procesos que quieran utilizar un periférico que ya esté asignado deben esperar a que sea liberado. Ello implica que durante períodos de mucha demanda varios procesos pueden quedar bloqueados a la espera del uso de unos determinados periféricos, mientras que durante otros períodos de tiempo puede que los mismos periféricos permanezcan sin ser utilizados. Con el fin de repartir la carga y reducir la posibilidad de cuellos de botella hay que adoptar algún otro tipo de estrategia.

La solución adoptada en varios sistemas consiste en recurrir al *spooling* de todas las E/S asociadas a periféricos muy utilizados. Ello significa que en lugar de efectuar la transferencia directamente al periférico asociado con un *stream*, la rutina de E/S lleva a cabo la transferencia sobre un soporte intermedio, normalmente sobre disco. La responsabilidad de transferir la información del disco al periférico en cuestión recae sobre un proceso independiente, denominado *spooler*, que está asociado con dicho periférico. Como ejemplo, consideremos un sistema en el que todas las salidas por impresora se hagan por *spooling*. A cada proceso que abra un *stream* asociado a la impresora se le asignará un fichero anónimo en disco, dirigiendo la rutina de E/S todas las salidas del *stream* hacia ese fichero que actuará como una impresora virtual. Cuando se cierre el *stream*, este fichero será añadido a una cola de ficheros similares creados por otros procesos, todos ellos esperando a ser impresos. La tarea del *spooler* de la impresora será la de ir cogiendo ficheros de esta cola y mandarlos a la impresora. Se supone, desde luego, que dentro de un período de tiempo determinado, la velocidad de esta impresora será suficiente como para imprimir todos los ficheros generados.



La estructura básica del *spooler* es la siguiente:

repetir de forma indefinida

inicio

wait (algo a imprimir);

sacar el fichero de la cola;

abrir el fichero;

repetir hasta fin de fichero

inicio

DOIO (parámetros para la lectura de disco);

wait (petición de disco servida);

DOIO (parámetros para la salida por impresora);

wait (petición de impresión servida);

final

final;

Se deben tener en cuenta los siguientes puntos:

1. Esta estructura se verá modificada en la práctica con el fin de permitir el *buffering* para los datos que vayan del disco a la impresora.
2. El semáforo *algo a imprimir* recibe un *signal* de cualquier proceso que cierre el *stream* de la impresora.
3. No hace falta que se sirva la cola de salida en base a un criterio del tipo "primer llegado, primer servido". El *spooler* puede favorecer por ejemplo, los ficheros cortos frente a los largos.

Las relaciones existentes entre el *spooler*, los gestores de periféricos y los procesos que generan la salida están esquematizadas en la Figura 30. Un diagrama similar, basado en análogas consideraciones, podría establecerse para un *spooler* de entrada.

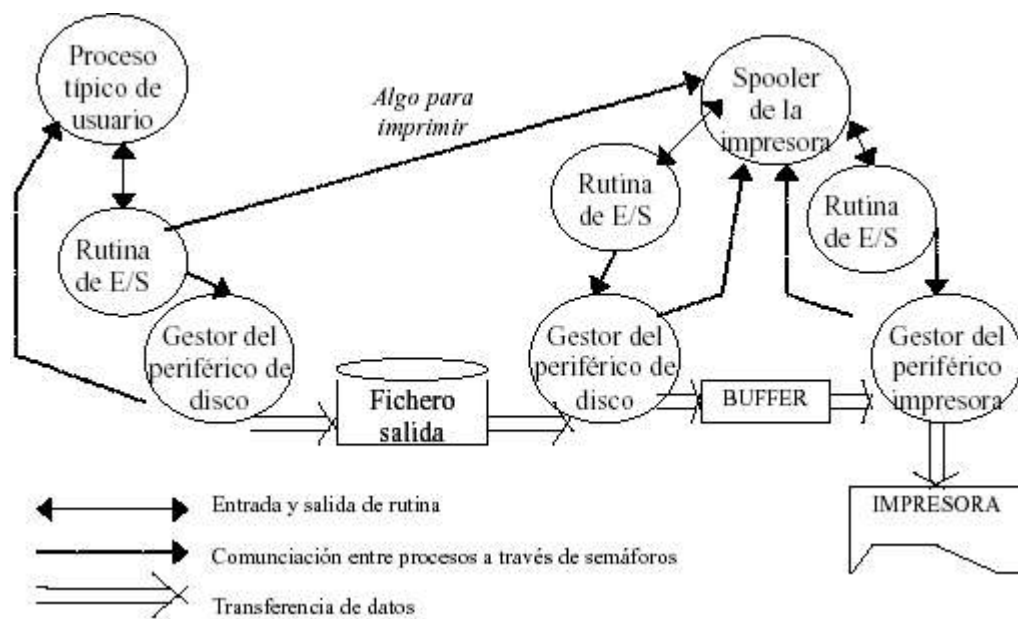


Figura 30. Salida por spooling.

En resumen, se puede afirmar que la técnica del *spooling* reduce la presión de la demanda en periféricos frecuentemente utilizados. También reduce la posibilidad de interbloqueos resultantes de asignaciones incorrectas de periféricos. Otra ventaja es que es bastante fácil obtener varias copias de la misma salida sin tener que volver a ejecutar la tarea. En contra hay que mencionar la gran cantidad de espacio de disco necesario para guardar las colas de entrada y de salida, así como el tráfico intenso que se produce en el canal del disco. Finalmente, decir que el *spooling* no es posible en entornos de tiempo real, ya que, en este caso, las E/S deben efectuarse de forma inmediata.

1.3.9 Compiladores.

¿Qué es un compilador?

Un traductor es cualquier programa que toma como entrada un texto escrito en un lenguaje, llamado fuente y da como salida otro texto en un lenguaje, denominado objeto.

Texto Lenguaje Fuente -----> Traductor -----> Texto Lenguaje Objeto

En el caso de que el lenguaje fuente sea un lenguaje de programación de alto nivel y el objeto sea un lenguaje de bajo nivel (ensamblador o código de máquina), a dicho traductor se le denomina compilador. Un ensamblador es un compilador cuyo lenguaje fuente es el lenguaje ensamblador. Un intérprete no genera un programa equivalente, sino que toma una sentencia del programa fuente en un lenguaje de alto nivel y la traduce al código equivalente y al mismo tiempo lo ejecuta.

Históricamente, con la escasez de memoria de los primeros ordenadores, se puso de moda el uso de intérpretes frente a los compiladores, pues el programa fuente sin traducir y el intérprete juntos daban una ocupación de memoria menor que la resultante de los compiladores. Por ello los primeros ordenadores personales iban siempre acompañados de un intérprete de BASIC (Spectrum, Commodore VIC-20, PC XT de IBM, etc.). La mejor información sobre los errores por parte del compilador así como una mayor velocidad de ejecución del código resultante hizo que poco a poco se impusieran los compiladores. Hoy en día, y con el problema de la memoria prácticamente resuelto, se puede hablar de un gran predominio de los compiladores frente a los intérpretes, aunque intérpretes como los incluidos en los navegadores de Internet para interpretar el código JVM de Java son la gran excepción.

Ventajas de compilar frente a interpretar

- Se compila una vez, se ejecuta n veces.
- En bucles, la compilación genera código equivalente al bucle, pero interpretándolo se traduce tantas veces una línea como veces se repite el bucle.
- El compilador tiene una visión global del programa, por lo que la información de mensajes de error es mas detallada.



Ventajas del intérprete frente al compilador

- Un intérprete necesita menos memoria que un compilador. En principio eran más abundantes dado que los ordenadores tenían poca memoria.
- Permiten una mayor interactividad con el código en tiempo de desarrollo.

Un compilador no es un programa que funciona de manera aislada, sino que necesita de otros programas para conseguir su objetivo: obtener un programa ejecutable a partir de un programa fuente en un lenguaje de alto nivel. Algunos de esos programas son el preprocesador, el linker (enlazador), el depurador y el ensamblador. El preprocesador se ocupa (dependiendo del lenguaje) de incluir ficheros, expandir macros, eliminar comentarios, y otras tareas similares. El linker se encarga de construir el fichero ejecutable añadiendo al fichero objeto generado por el compilador las cabeceras necesarias y las funciones de librería utilizadas por el programa fuente. El depurador permite, si el compilador ha generado adecuadamente el programa objeto, seguir paso a paso la ejecución de un programa. Finalmente, muchos compiladores, en vez de generar código objeto, generan un programa en lenguaje ensamblador que debe después convertirse en un ejecutable mediante un programa ensamblador.

Clasificación de Compiladores

El programa compilador traduce las instrucciones en un lenguaje de alto nivel a instrucciones que la computadora puede interpretar y ejecutar. Para cada lenguaje de programación se requiere un compilador separado. El compilador traduce todo el programa antes de ejecutarlo. Los compiladores son, pues, programas de traducción insertados en la memoria por el sistema operativo para convertir programas de cómputo en pulsaciones electrónicas ejecutables (lenguaje de máquina). Los compiladores pueden ser de:

- Una sola pasada: examina el código fuente una vez, generando el código o programa objeto.
- Pasadas múltiples: requieren pasos intermedios para producir un código en otro lenguaje, y una pasada final para producir y optimizar el código producido durante los pasos anteriores.
- Optimización: lee un código fuente, lo analiza y descubre errores potenciales sin ejecutar el programa.
- Compiladores incrementales: generan un código objeto instrucción por instrucción (en vez de hacerlo para todo el programa) cuando el usuario teclea cada orden individual. El otro tipo de compiladores requiere que todos los enunciados o instrucciones se compilen conjuntamente.
- Ensamblador: el lenguaje fuente es lenguaje ensamblador y posee una estructura sencilla.



- **Compilador cruzado:** se genera código en lenguaje objeto para una máquina diferente de la que se está utilizando para compilar. Es perfectamente normal construir un compilador de Pascal que genere código para MS-DOS y que el compilador funcione en Linux y se haya escrito en C++.
- **Compilador con montador:** compilador que compila distintos módulos de forma independiente y después es capaz de enlazarlos.
- **Autocompilador:** compilador que está escrito en el mismo lenguaje que va a compilar. Evidentemente, no se puede ejecutar la primera vez. Sirve para hacer ampliaciones al lenguaje, mejorar el código generado, etc.
- **Metacompilador:** es sinónimo de compilador de compiladores y se refiere a un programa que recibe como entrada las especificaciones del lenguaje para el que se desea obtener un compilador y genera como salida el compilador para ese lenguaje. El desarrollo de los metacompiladores se encuentra con la dificultad de unir la generación de código con la parte de análisis. Lo que sí se han desarrollado son generadores de analizadores léxicos y sintácticos. Por ejemplo, los conocidos:
 - **LEX:** generador de analizadores léxicos
 - **YACC:** generador de analizadores sintácticos desarrollados para UNIX. Los inconvenientes que tienen son que los analizadores que generan no son muy eficientes.
- **Descompilador:** es un programa que acepta como entrada código máquina y lo traduce a un lenguaje de alto nivel, realizando el proceso inverso a la compilación.

Funciones de un compilador

A grandes rasgos un compilador es un programa que lee un programa escrito en un lenguaje, el lenguaje fuente, y lo traduce a un programa equivalente en otro lenguaje, el lenguaje objeto. Como parte importante de este proceso de traducción, el compilador informa a su usuario de la presencia de errores en el programa fuente.

A primera vista, la diversidad de compiladores puede parecer abrumadora. Hay miles de lenguajes fuente, desde los lenguajes de programación tradicionales, como FORTRAN o Pascal, hasta los lenguajes especializados que han surgido virtualmente en todas las áreas de aplicación de la informática. Los lenguajes objeto son igualmente variados; un lenguaje objeto puede ser otro lenguaje de programación o el lenguaje de máquina de cualquier computador entre un microprocesador y un supercomputador. A pesar de existir una aparente complejidad por la clasificación de los compiladores, como se vio en el tema anterior, las tareas básicas que debe realizar cualquier compilador son esencialmente las mismas. Al comprender tales tareas, se pueden construir compiladores para una gran diversidad de lenguajes fuente y máquinas objeto utilizando las mismas técnicas básicas.

Nuestro conocimiento sobre cómo organizar y escribir compiladores ha aumentado mucho desde que comenzaron a aparecer los primeros compiladores a principios de los años cincuenta. Es difícil dar una fecha exacta de la aparición del primer compilador, porque en un principio gran parte del trabajo de experimentación y aplicación se realizó de manera independiente por varios grupos. Gran parte de los primeros trabajos de compilación estaba relacionada con la traducción de fórmulas aritméticas a código de máquina. En la década de 1950, se consideró a los compiladores como programas notablemente difíciles de escribir. EL primer compilador de FORTRAN, por ejemplo, necesitó para su implantación de 18 años de trabajo en grupo (Backus y otros [1975]). Desde entonces, se han descubierto técnicas sistemáticas para manejar muchas de las importantes tareas que surgen en la compilación. También se han desarrollado buenos lenguajes de implantación, entornos de programación y herramientas de software. Con estos avances, puede hacerse un compilador real incluso como proyecto de estudio en un curso de un semestre sobre diseño sobre de compiladores.

Partes en las que trabaja un compilador

Conceptualmente un compilador opera en fases. Cada una de las cuales transforma el programa fuente de una representación en otra. En la figura siguiente se muestra una descomposición típica de un compilador. En la práctica se pueden agrupar fases y las representaciones intermedias entre las fases agrupadas no necesitan ser construidas explícitamente.

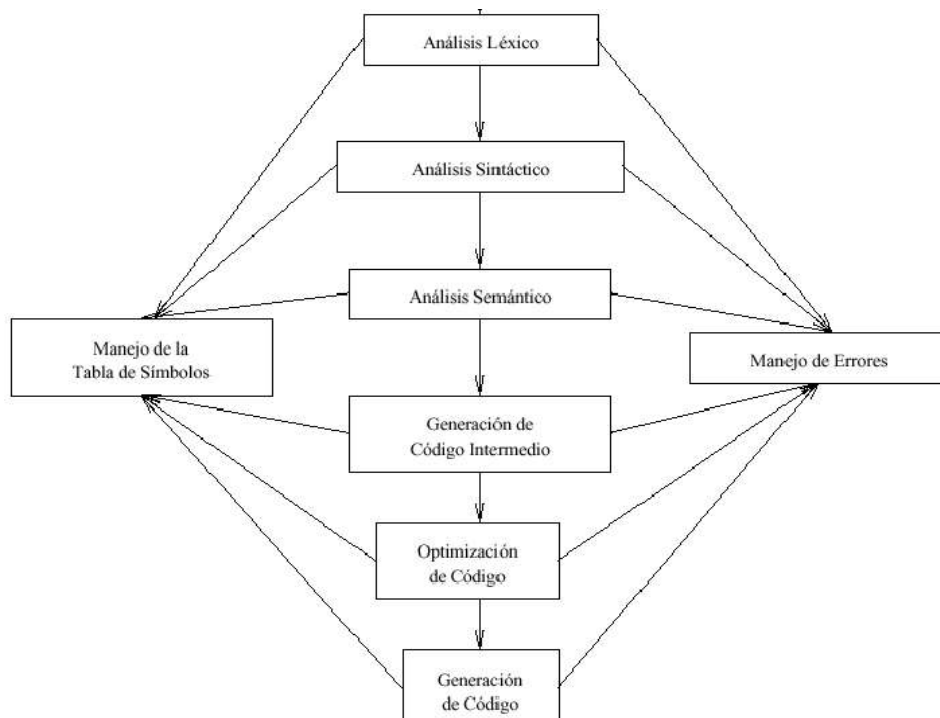


Figura 31. Partes del compilador

Para más información consultar el anexo correspondiente a compiladores.