Índice

1 Intro:

Hace muchos años se hizo muy evidente que debía encontrarse alguna forma de proteger a los programadores de la complejidad del hardware. La solución que ha evolucionado gradualmente consiste en poner una capa de software encima del hardware solo, que se encargue de administrar todas las partes del sistema y presente al usuario una interfaz o **máquina virtual** que sea más fácil de entender y programar. Esta capa de software es el sistema operativo.

En algunas máquinas viene una capa de software primitivo que controla directamente estos dispositivos y ofrece una interfaz más aseada a la siguiente capa. Este software, llamado **microprograma,** suele estar almacenado en memoria de sólo lectura. En realidad es un intérprete, que obtiene las instrucciones de lenguaje de máquina como ADD, MOVE y JUMP y las ejecuta en una serie de pasos pequeños. Por ejemplo, para ejecutar una instrucción ADD (sumar), el microprograma debe determinar dónde se encuentran los números que se van a sumar, obtenerlos, sumarlos y almacenar el resultado en algún lugar. El conjunto de instrucciones que el microprograma interpreta define el **lenguaje de máquina**, que no es realmente parte de la máquina

física, aunque los fabricantes de computadoras siempre lo describen en sus manuales como tal, de modo que muchas personas piensan en él como si fuera la "máquina" real.

Algunas computadoras, llamadas **RISC (computadoras con conjunto de instrucciones reducido)**, no tienen un nivel de microprogramación. En estas máquinas, el hardware ejecuta las instrucciones de lenguaje de máquina directamente. Por ejemplo, el Motorola 680x0 tiene un nivel de microprogramación, pero el IBM PowerPC no.

Una función importante del sistema operativo es ocultar toda esta complejidad y ofrecer al programador un conjunto de instrucciones más cómodo con el que pueda trabajar. Por ejemplo, LEER BLOQUE DE ARCHIVO es conceptualmente más sencillo que tener que preocuparse por los detalles de mover cabezas de disco, esperar que se estabilicen, etcétera.

Encima del sistema operativo está el resto del software de sistema. Aquí encontramos el intérprete de comandos (shell), sistemas de ventanas, compiladores, editores y otros programas similares independientes de la aplicación. Es importante darse cuenta de que estos programas definitivamente no forman parte del sistema operativo, a pesar de que casi siempre son provistos por el fabricante de la computadora. Éste es un punto crucial, aunque sutil. El sistema operativo es la porción del software que se ejecuta en **modo kernel o modo supervisor**, y está protegido por el hardware contra la intervención del usuario (olvidándonos por el momento de algunos de los microprocesadores más viejos que no tienen ninguna protección de hardware). Los compiladores y editores se ejecutan en **modo de usuario**. Si a un usuario no le gusta un compilador en particular, él+ está en libertad de escribir el suyo propio si lo desea; no está en libertad de escribir su propio manejador de interrupciones del disco, que forma parte del
sistema operativo y normalmente está protegido por el hardware contra los intentos de los usuarios por modificarlo.

El programa que oculta la verdad acerca del hardware y presenta al programador una vista sencilla y bonita de archivos con nombre que pueden leerse y escribirse es, por supuesto, el sistema operativo. Así como el sistema operativo aísla al programador del hardware del disco y presenta una interfaz sencilla orientada a archivos, también oculta muchos asuntos desagradables referentes a interrupciones, temporizadores, administración de memoria y otras funciones de bajo nivel, En cada caso, la abstracción que el sistema operativo ofrece es más sencilla y fácil de usar que el hardware subyacente. En esta vista, la función del sistema operativo es presentar al usuario el equivalente de una **máquina extendida o máquina virtual** que es más fácil de programar que el hardware subyacente.

2 Tipos de sistemas operativos:

Define the essential properties of the following types of operating systems:
a. Batch
b. Interactive
c. Time sharing
d. Real time
e. Network
f. Distributed

Answer:
a. **Batch**. Jobs with similar needs are batched together and run through the computer as a group by an operator or automatic job sequencer. Performance is increased by attempting to keep CPU and I/O devices busy at all times through buffering, off-line operation, spooling, and multiprogramming. Batch is good for executing large jobs that need little interaction; it can be submitted and picked up later.
b. **Interactive**. This system is composed of many short transactions where the results of the next transactionmay be unpredictable. Response time needs to be short (seconds) since the user submits and waits for the result.
c. **Time sharing**. This systems uses CPU scheduling and multiprogramming to provide economical interactive use of a system. The CPU switches rapidly from one user to another. Instead of having a job defined by spooled card images, each program reads
d. **Real time**. Often used in a dedicated application, this system reads information from sensors and must respond within a fixed amount of time to ensure correct performance.
e. **Network**.
f. **Distributed**.This system distributes computation among several physical processors. The processors do not share memory or a clock. Instead, each processor has its own local memory. They communicate with each other through various communication lines, such as a high-speed bus or telephone line.

We have stressed the need for an operating system to make efficient use of the computing hardware. When is it appropriate for the operating system to forsake this principle and to "waste" resources? Why is such a system not really wasteful?
Answer: Single-user systems should maximize use of the system for the user. A GUI might "waste" CPU cycles, but it optimizes the user's interaction with the system.

3 Multiprogramacion:

Aún con el uso de interrupciones, puede que un procesador no esté aprovechado de una manera muy eficiente. Por ejemplo, considérese de nuevo la figura 1.9b. Si el tiempo nece- sario para completar una operación de E/S es mucho mayor que el código del usuario entre llamadas de E/S (una situación habitual), entonces el procesador va a estar desocupado du- rante gran parte del tiempo. Una solución a este problema es permitir que varios programas de usuario estén activos a un mismo tiempo.

Cuando el procesador tiene que tratar con una serie de programas, la secuencia en que es- tos se ejecutan dependerá de su prioridad relativa y de si están esperando una E/S. Cuando un programa es interrumpido y se transfiere el control a la rutina de tratamiento de la inte- rrupción, una vez que ésta haya terminado, puede que no se devuelva el control inmediata- mente al programa de usuario que estaba ejecutándose en el momento de la interrupción. En su lugar, el control puede transferirse a algún otro programa pendiente que tenga mayor prioridad. Finalmente, cuando tenga la prioridad más alta, se reanudará el programa de usuario que fue interrumpido. Esta situación de varios programas que se ejecutan por turnos se conoce como *multiprogramación*

What is the main advantage of multiprogramming?
Answer: Multiprogramming makes efficient use of the CPU by overlapping the demands for the CPU and its I/O devices from various users. It attempts to increase CPU utilization by always having something for the CPU to execute.

What are the main differences between operating systems for mainframe computers and personal computers?
Answer: The design goals of operating systems for those machines are quite different. PCs are inexpensive, so wasted resources like CPU cycles are inconsequential. Resources are wasted to improve usability and increase software user interface functionality. Mainframes are the opposite, so resource use is maximized, at the expensive of ease of use.

In a multiprogramming and time-sharing environment, several users share the system simultaneously.
This situation can result in various security problems.
a. What are two such problems?
b. Can we ensure the same degree of security in a time-shared machine as we have in a dedicated machine? Explain your answer.
Answer:
a. Stealing or copying one's programs or data; using system resources (CPU, memory, disk space, peripherals) without proper accounting.
b. Probably not, since any protection scheme devised by humans can inevitably be broken by a human, and the more complex the scheme, the more difficult it is to feel confident of its correct implementation.

4 Controladores:

-Comparten el bus
-Cada uno se encarga de un tipo especifico de dispositivo, pudiendo haber varios del mismo tipo conectados al mismo controlador.
-Un controlador mantiene un buffer local y un conjunto de registros de propósito especial; El controlador se encarga de transferir los datos entre los dispositivos periféricos que controla y su buffer local.

5 Operación en modo dual:

How does the distinction between monitor mode and user mode function as a rudimentary form of protection (security) system?
Answer: By establishing a set of privileged instructions that can be executed only when in the monitor mode, the operating system is assured of controlling the entire system at all times.

Some computer systems do not provide a privileged mode of operation in hardware. Consider

whether it is possible to construct a secure operating system for these computers.
Give arguments both that it is and that it is not possible.
Answer: An operating system for a machine of this type would need to remain in control (or monitor mode) at all times. This could be accomplished by two methods:
a. Software interpretation of all user programs (like some BASIC, APL, and LISP systems, for example). The software interpreter would provide, in software, what the hardware does not provide.
b. Require meant that all programs be written in high-level languages so that all object code is compiler-produced. The compiler would generate (either in-line or by function calls) the protection checks that the hardware is missing.

6 Protección:

Protecting the operating system is crucial to ensuring that the computer system operates correctly. Provision of this protection is the reason behind dual-mode operation, memory protection, and the timer. To allow maximum flexibility, however, we would also like to place minimal constraints on the user.
The following is a list of operations that are normally protected. What is the *minimal* set of instructions that must be protected?

a. Change to user mode.
b. Change to monitor mode.
c. Read from monitor memory.
d. Write into monitor memory.
e. Fetch an instruction from monitor memory.
f. Turn on timer interrupt.
g. Turn off timer interrupt.
Answer: The minimal set of instructions that must be protected are:
a. Change to monitor mode.
b. Read from monitor memory.
c. Write into monitor memory.
d. Turn off timer interrupt.

Some early computers protected the operating system by placing it in a memory partition that could not be modified by either the user job or the operating system itself. Describe two difficulties that you think could arise with such a scheme.
Answer: The data required by the operating system (passwords, access controls, accounting information, and so on) would have to be stored in or passed through unprotected memory and thus be accessible to unauthorized users.

Writing an operating system that can operate without interference from malicious or undebugged user programs requires some hardware assistance. Name three hardware aids for writing an operating system, and describe how they could be used together to protect the operating system.
Answer:
a. Monitor/user mode
b. Privileged instructions
c. Timer
d. Memory protection

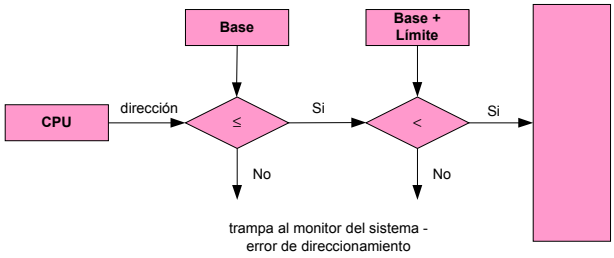
6.1.1 Protección de E/S

Para evitar que un usuario realice E/S no válida, todas las instrucciones de E/S son privilegiadas, los usuarios deben ceder el control al sistema operativo. Si puedo acceder a la memoria, al vector de interrupciones, y alterar la dirección de la interrupción que genera el hardware, puedo asumir el control del computador en modo privilegiado, para que esto no ocurra, también debo proteger la memoria.

6.1.2 Protección de la memoria

Debemos impedir que el vector de interrupción sea modificado por un programa de usuario, al igual que las rutinas de servicio de interrupciones del SO. Esta protección debe correr por cuenta del hardware.

Podemos separar el espacio de memoria de cada programa, determinando el espacio de direcciones, al cual el programa puede acceder, y proteger el resto de la memoria que no está en ese espacio. Se almacena en registros, el rango de memoria privilegiada, y se efectúa interrupción si cualquier programa en modo usuario, intenta acceder a esa zona de memoria.



Los registros de control son:

- **Registro base:** Contiene la dirección de memoria física, válida más pequeña.
- **Registro límite:** Contiene el tamaño del intervalo.

El hardware de la CPU, compara todas las direcciones generadas por el usuario, con estos registros, pasando el control al SO, si se produce algún error. El SO tiene acceso irrestricto a la memoria, pudiendo cargar los programas usuario en cualquier lado, y pudiendo expulsar esos programas en caso de error.

6.1.3 Protección de la CPU

Debemos impedir que un programa de usuario se atasque en un loop y nunca devuelva el control al sistema operativo. Para ello, existe un *timer*, que interrumpe al computador después de un período determinado, y cede el control al SO, que puede cerrar el programa o cederle más tiempo. Las instrucciones que modifican el funcionamiento del timer, son privilegiadas.

Otra implementación de los timers es para sistemas de tiempo compartido, dándole un tiempo determinado para los programas de cada usuario; o para el cálculo de la hora actual.

7 Caches:

Aunque la memoria caché es invisible para el sistema operativo, interactúa con otras partes del hardware de gestión de memoria. Es más, muchos de los principios utilizados en la me- moria virtual son también aplicables a la memoria caché.

Motivación
En todos los ciclos de instrucción, el procesador accede a la memoria, al menos una vez, para leer la instrucción y, a menudo, algunas veces más para leer los operandos y/o almacenar los resultados. La frecuencia con que el procesador puede ejecutar las instrucciones está claramente limitada por el tiempo de ciclo de memoria. Esta limitación ha sido de hecho un problema significativo, debido al persistente desacuerdo entre las velocidades del procesador y de la memoria principal. La figura 1.17, que puede considerarse representativa, ilustra esta situación.

La figura muestra que la velocidad de la memoria no se corresponde con la velocidad del procesador. Se debe adoptar un compromiso entre la velocidad, el coste y el tamaño de la memoria. En el mejor de los casos, la memoria principal se construiría con la misma tecno-

logía que los registros del procesador, obteniéndose unos tiempos de ciclo de memoria com-parables con los de los ciclos del procesador. Esta estrategia siempre ha resultado demasiado costosa. La solución es aprovechar el principio de cercanía, disponiendo una memoria pequeña y rápida entre el procesador y la memoria principal, es decir, la caché.

Principios de la cache
La memoria caché intenta obtener una velocidad cercana a la de las memorias más rápidas y, al mismo tiempo, aportar una memoria grande al precio de las memorias de semiconductores, que son menos costosas. Este concepto se ilustra en la figura 1.18. Hay una memoria principal más lenta y relativamente grande, junto a una memoria caché más pequeña y rápida. La caché contiene una copia de parte de la memoria principal. Cuando el procesador intenta leer una palabra de la memoria, se comprueba si la palabra está en la memoria caché. Si es así, la palabra se envía al procesador. Si no, se rellena la caché con un bloque de memoria principal, formado por un número fijo de palabras y, después, la palabra es enviada al procesador. Debido al fenómeno de la cercanía de referencias, cuando se carga en la caché un bloque de datos para satisfacer una sola referencia a memoria, es probable que ya se hayan hecho antes otras referencias a palabras del mismo bloque.

La figura 1.19 representa la estructura de un sistema de caché y memoria principal. La memo-ria principal consta de hasta 2^n palabras direccionables, teniendo cada palabra una única direc-ción de n bits. Se considera que esta memoria consta de un número de bloques de longitud fija de K palabras cada uno. Es decir, hay $M = 2^n/K$ bloques. La caché consta de C secciones de K palabras cada una y el número de secciones es considerablemente menor que el número de bloques de memoria principal $(C«M)$. En todo momento, algún subconjunto de los bloques de memoria reside en las secciones de la caché. Si se va a leer una palabra de un bloque de memoria que no está en caché, entonces el bloque se

transfiere para alguna de las secciones de la caché. Debido a que hay más bloques que secciones, una sección individual no puede ser dedicada única y permanentemente a un bloque en particular. De este modo, cada sección incluye un indicador que identifica cuál es el bloque particular que está siendo actualmente almacenado en ella. Este indicador es usualmente un cierto número de los bits más significativos de la dirección.

La figura 1.20 ilustra la operación de LEER. El procesador genera la dirección *DL,* de la palabra a leer. Si la palabra está en la cache es entregada al procesador. En caso contrario, el bloque que contiene a la palabra se carga en cache y la palabra se envía al procesador.

Diseño de la cache

Una discusión detallada sobre la estructura de la cache se escapa del alcance de este libro. Aquí se resumirán de forma breve los elementos clave. Se comprobará que hay que abordar cuestiones similares cuando se haga frente al diseño de la memoria virtual y de la cache del disco. Estos aspectos se encuadran en las siguientes categorías:

1• Tamaño de cache.
2• Tamaño del bloque.
3• Función de correspondencia (*mapping*)
4• Algoritmo de reemplazo.
5• Política de escritura.

Ya se ha hablado sobre el **tamaño de cache**. De aquí se desprende que caches razonaclemente pequeñas pueden tener un impacto significativo sobre el rendimiento. Otra cuestión de tamaño es el **tamaño del bloque,** que es la unidad de intercambio de datos entre la cache y la memoria principal. A medida que el tamaño del bloque aumenta, desde los blo- ques más pequeños a los más grandes, la tasa de aciertos (proporción de veces que una re- ferencia se encuentre en la cache) aumentará al comienzo, debido al principio de cercanía: la alta probabilidad de que se haga referencia en un futuro próximo a datos cercanos a la palabra referenciada. A medida en que el tamaño del bloque crece, pasan a la cache más da- tos útiles. Sin embargo, la tasa de aciertos comenzará a disminuir, dado que el bloque se hace aún mayor y la probabilidad de uso del dato leído más recientemente se hace menor que la probabilidad de reutilizar el dato que hay que sacar de la cache para hacer sitio al bloque nuevo.

Cuando se trae a cache un nuevo bloque de datos, la **función de correspondencia** *(map-ping)* determina la posición de la cache que ocupará el bloque. Dos limitaciones influyen en el diseño de la función de traducción. En primer lugar, cuando un bloque se trae a la cache, puede que otro tenga que ser reemplazado. Convendría hacer esto de forma que se redujera la probabilidad de reemplazar un bloque que se vaya a necesitar en un futuro próximo.

Mientras más flexible sea la función de traducción, mayor será la envergadura del diseño del algoritmo de reemplazo que maximice la tasa de aciertos. En segundo lugar, cuanto más flexible sea la función de traducción, más compleja será la circuitería necesaria para determinar si un bloque dado está en la cache.

El **algoritmo de reemplazo** escoge, bajo las restricciones de la función de traducción, el bloque que hay que reemplazar: Sena conveniente reemplazar aquel bloque que tenga me- nos probabilidad de necesitarse en un futuro cercano. Aunque es imposible identificar un bloque tal, una estrategia bastante efectiva es reemplazar el bloque que lleva más tiempo en la cache sin que se hayan hecho referencias a él. Esta política se denomina algoritmo del *usado hace más tiempo* (LRU, *Least Recently Used).* Se necesitan mecanismos de hardware para identificar el bloque usado hace más tiempo.

Si se modifica el contenido de un bloque de la cache, entonces hace falta escribirlo de nuevo a la memoria principal, antes de reemplazarlo. La **política de escritura** dicta cuándo tiene lugar la operación de escribir en memoria. Por un lado, la escritura puede producirse cada vez que el bloque se actualice. Por otro lado, la escritura se produce sólo cuando se reemplace el bloque. Esta última política reduce las operaciones de escritura en memoria pero deja la memoria principal en un estado obsoleto. Esto puede dificultar la operación de los multiprocesadores y el acceso directo a memoria por parte de los módulos de E/S.

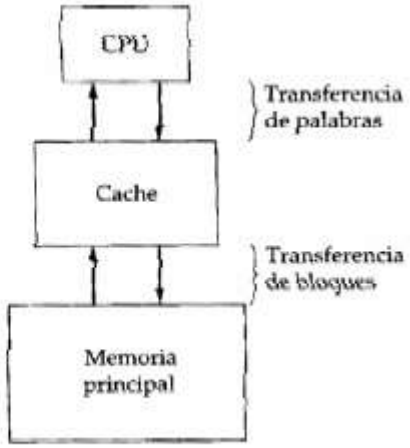


FIGURA 1.18 Cache y memoria principal

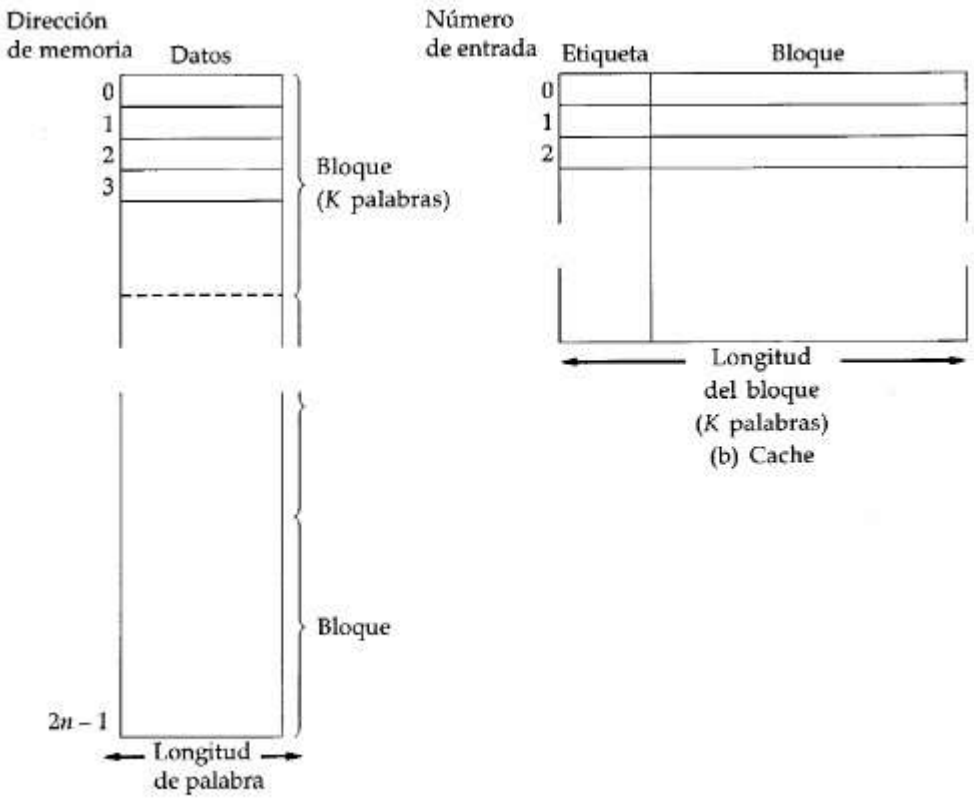


FIGURA 1.19 Estructura de cache/memoria principal

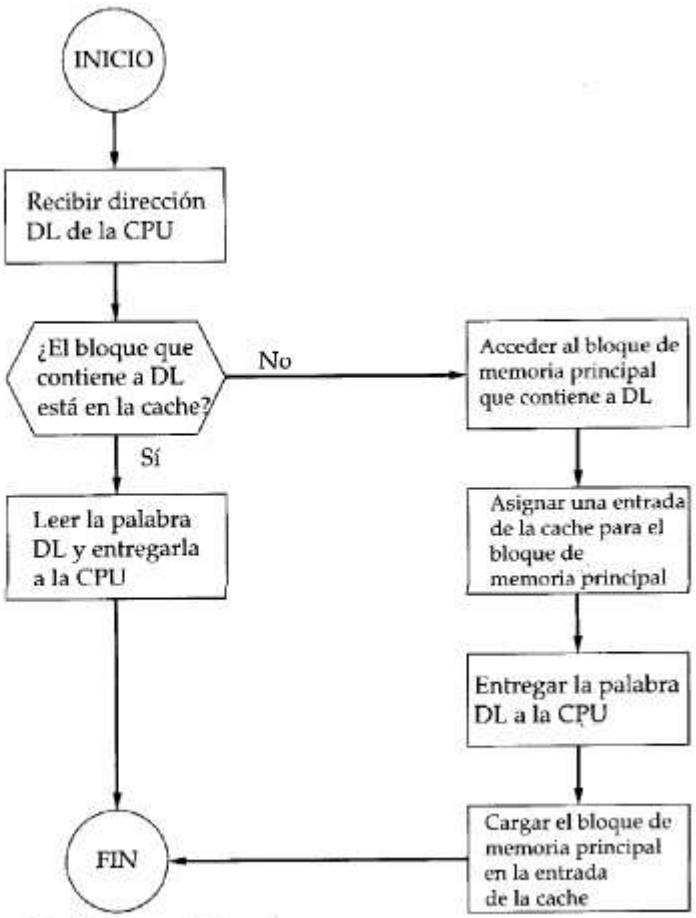


FIGURA 1.20 Operación de lectura de la cache

Give two reasons why caches are useful. What problems do they solve? What problems do they cause? If a cache can be made as large as the device for which it is caching (for instance, a cache as large as a disk), why not make it that large and eliminate the device?
Answer: Caches are useful when two or more components need to exchange data, and the components perform transfers at differing speeds. Cahces solve the transfer problem by providing a buffer of intermediate speed between the components. If the fast device finds the data it needs in the cache, it need not wait for the slower device. The data in the cache must be kept consistent with the data in the components. If a component has a data value change, and the datum is also in the cache, the cache must also be updated. This is especially a problem on multiprocessor systems where more than one process may be accessing a datum. A component may be eliminated by an equal-sized cache, but only if: (a) the cache and the component have equivalent state-saving capacity (that is, if the component retains its data when electricity is removed, the cache must retain data as well), and (b) the cache is affordable, because faster storage tends to be more expensive.

8 Llamadas al sistema:

Son la interfaz entre un proceso y el sistema operativo, y están generalmente disponibles como instrucciones en lenguaje ensamblador, aunque algunos sistemas permiten emitir llamadas al sistema desde un programa escrito en lenguaje de alto nivel.

What is the purpose of system programs?
Answer: System programs can be thought of as bundles of useful system calls. They provide basic functionality to users and so users do not need to write their own programs to solve common problems.

What is the purpose of system calls?
Answer: System calls allow user-level processes to request services of the operating system.
Una llamada al sistema cambia de modo usuario a modo núcleo.

9 Funciones de E/S :

[sta]

Hasta ahora se ha discutido que el funcionamiento del computador es controlado por el procesador y se ha analizado en primer lugar la interacción entre el procesador y la memoria. En la discusión sólo se ha aludido al papel de la componente de E/S. Los módulos de E/S (un controlador de disco, por ejemplo) pueden intercambiar datos di-rectamente con el procesador. Al igual que el procesador puede iniciar una lectura o escritura en la memoria, indicando la dirección de una ubicación específica, el procesador también puede leer datos de un módulo de E/S o escribir datos en el módulo. En este último caso, el procesador identifica a un dispositivo específico que es controlado por un módulo de E/S determinado. De este modo se podría tener una secuencia de instrucciones similar a la de la figura 1.4, pero con instrucciones de E/S en lugar de referencias a memoria. En algunos casos, es conveniente permitir que los intercambios de E/S se produzcan directamente con la memoria. En tal caso, el procesador dará autoridad a un módulo de E/S para leer o escribir en memoria, de modo que la transferencia de E/S ocurre sin obstruir al procesador. Durante la transferencia, el módulo de E/S emite órdenes de lectura o escritura en la memoria, librando de responsabilidades al procesador en el intercambio. Esta operación se conoce como *acceso directo a memoria* (DMA, *Direct Memory Access)* y será examinada más adelante en este mismo capítulo. Por ahora, todo lo que se necesita saber es que la estructura de interconexión del computador puede que tenga que permitir una interacción directa entre la memoria y la E/S.

10 TÉCNICAS DE COMUNICACIÓN DE E/S

Para las operaciones de E/S son posibles las tres técnicas siguientes:

- E/S programada
- E/S dirigida por interrupciones
- Acceso Directo a Memoria (DMA: Direct Memory Access)

10.1.1 E/S programada

Cuando el procesador está ejecutando un programa y encuentra una instrucción de E/S, ejecuta dicha instrucción, enviando una orden al módulo apropiado de E/S. Con E/S programada, el módulo de E/S llevará a cabo la acción requerida y luego activará los bits

apropiados en el registro de estado de E/S. El módulo de E/S no lleva a cabo ninguna otra acción para avisar al procesador. En particular, no interrumpe al procesador. Así pues, es responsabilidad del procesador comprobar periódicamente el estado del módulo de E/S hasta saber que se ha completado la operación.

Con esta técnica, el procesador es el responsable de extraer los datos de la memoria principal cuando va a hacer una salida o poner los datos en la memoria principal cuando se hace una entrada. El software de E/S se escribe de manera tal que el procesador ejecute unas instrucciones que le otorguen el control directo sobre la operación de E/S, incluyendo la comprobación del estado de los dispositivos, el envío de órdenes de lectura o escritura y la transferencia de los datos. Por lo tanto, en el conjunto de instrucciones se incluyen instrucciones de E/S de las siguientes categorías:

- *Control:* Empleadas para activar un dispositivo externo y decirle qué debe hacer. Por ejemplo, una unidad de cinta magnética puede ser instruida para rebobinar o avanzar un registro.
- *Comprobación:* Empleadas para comprobar varias condiciones de estado asociadas con un módulo de E/S y sus periféricos.
- *Lectura, escritura:* Empleadas para transferir los datos entre los registros del procesador y los dispositivos externos.

La figura 1.21a muestra un ejemplo de utilización de la E/S programada para leer en un bloque de datos desde un dispositivo externo (por ejemplo, un registro de una cinta) hacia la memoria. Los datos se leen por palabra (por ejemplo, 16 bits) cada vez. Para cada palabra que se lea, el procesador debe permanecer en un bucle de comprobación del estado hasta que determine que la palabra está disponible en el registro de datos del módulo de E/S. El diagrama que se muestra en la figura 1.21 destaca la desventaja principal de esta técnica: Es un proceso que consume tiempo y que mantiene al procesador ocupado de forma innecesaria.

10.1.2 E/S dirigida por interrupciones

El problema de la E/S programada es que el procesador tiene que esperar un largo rato a que el módulo de E/S en cuestión esté listo para recibir o transmitir más datos. El procesador, mientras está esperando, debe interrogar repetidamente el estado del módulo de E/S. Como resultado, el nivel de rendimiento del sistema en conjunto se degrada fuertemente.

Una alternativa es que el procesador envíe una orden de E/S al módulo y se dedique a ha- cer alguna otra tarea útil. El módulo de E/S interrumpirá entonces al procesador para reque- rir sus servicios cuando esté listo para intercambiar los datos. El procesador ejecuta enton- ces la transferencia de los datos y reanuda el procesamiento anterior.

Seguidamente veremos cómo funciona esto, en primer lugar desde el punto de vista del módulo de E/S. Para la entrada, el módulo de E/S recibe una orden LEER desde el procesador. El módulo de E/S procede entonces con la lectura de los datos desde el periférico asociado. Una vez que los datos están en el registro de datos del módulo, éste envía una señal de interrupción al procesador a través de una línea de control. El módulo espera entonces a que los datos sean solicitados por el procesador. Cuando se haga esta solicitud, el módulo pondrá los datos en el bus de datos y estará listo para otra operación de E/S.

Desde el punto de vista del procesador, la acción para la entrada es como sigue. El procesador envía una orden LEER. Salva entonces el contexto (el contador de programa y los registros del procesador, por ejemplo) del programa en curso, se sale del mismo y se dedica a hacer otra cosa (por ejemplo, el procesador puede estar trabajando con diferentes programas al mismo tiempo). Al finalizar cada ciclo de instrucción, el procesador comprueba si hubo alguna interrupción (figura 1.7). Cuando se produce una interrupción desde el módulo de E/S, el procesador salva el contexto del programa que está ejecutando en ese momento y comienza la ejecución de la rutina de tratamiento de la interrupción. En este caso, el procesador lee la palabra de datos del módulo de E/S y la almacena en la memoria. Luego restaura el contexto del programa que emitió la orden de E/S (o de algún otro programa) y reanuda la ejecución.

La figura 1.2 Ib muestra el uso de E/S por interrupciones para leer en un bloque de datos, Compárese con la figura 1.2ª, La E/S por interrupciones es más eficiente que la E/S programada porque elimina las esperas innecesarias. Sin embargo, la E/S por interrupciones sigue

consumiendo una gran cantidad de tiempo del procesador, debido a que cada palabra de datos que va de la memoria al módulo de E/S o del módulo de E/S a la memoria debe pasar a través del procesador.

Casi siempre habrá varios módulos de E/S en un sistema informático, así que hacen falta mecanismos que capaciten al procesador para determinar qué dispositivo causó la interrupción y decidir, en caso de varias líneas de interrupción, cuál debe tratar primero. En algunos sistemas, hay varias líneas de interrupción, de forma que cada módulo de E/S envía una señal por una línea diferente. Cada línea tiene una prioridad diferente. Otra solución sería habilitar una única línea de interrupción, pero utilizando líneas adicionales para indicar la dirección del dispositivo. De nuevo, a diferentes dispositivos se les asignarán prioridades diferentes.

What are the differences between a trap and an interrupt? What is the use of each function?
Answer: An interrupt is a hardware-generated change-of-flow within the system. An interrupt handler is summoned to deal with the cause of the interrupt; control is then returned to the interrupted context and instruction. A trap is a software-generated interrupt. An interrupt can be used to signal the completion of an I/O to obviate the need for device polling. A trap can be used to call operating system routines or to catch arithmetic errors.

10.1.3 Acceso directo a memoria

La E/S dirigida por interrupciones, aunque es más eficiente que la simple E/S programada, todavía requiere de la intervención activa del procesador para transferir los datos entre la memoria y un módulo de E/S y, además, cualquier transferencia de datos debe recorrer un camino que pasa por el procesador. Así pues, ambas formas de E/S adolecen de dos desventajas inherentes:

1. La velocidad de transferencia de E/S está limitada por la velocidad con la que el procesador puede comprobar y dar servicio a un dispositivo.
2. El procesador participa en la gestión de la transferencia de E/S; debe ejecutarse una se- rie de instrucciones en cada transferencia de E/S.

Cuando se tienen que mover grandes volúmenes de datos, se necesita una técnica más efi-ciente: el acceso directo a memoria (DMA, *Direct Memory Access)*. La función de DMA se puede llevar a cabo por medio de un módulo separado sobre el bus del sistema o puede estar incorporada dentro de un módulo de E/S. En cualquier caso, la técnica funciona como sigue. Cuando el procesador desea leer o escribir un bloque de datos, emite una orden hacia el módulo de DMA, enviándole la información siguiente:

- Si lo que se solicita es una lectura o una escritura
- La dirección del dispositivo de E/S involucrado
- La dirección inicial de memoria desde la que se va a leer o a la que se va a escribir
- El número de palabras a leer o escribir

El procesador continúa entonces con otro trabajo. Habrá delegado la operación de E/S en el módulo de DMA y dicho módulo es el que tendrá que encargarse de ésta. El módulo de DMA transfiere el bloque entero, una palabra cada vez, directamente hacia o desde la memoria, sin pasar por el procesador. Cuando se completa la transferencia, el módulo de DMA envía una señal de interrupción al procesador. De esta manera, el procesador se ve involucrado sólo al inicio y al final de la transferencia (figura 1.21c).

El módulo de DMA debe tomar el control del bus para transferir los datos con la memoria. Debido a la competencia por la utilización del bus, puede ocurrir que el procesador necesite el bus pero deba esperar. Nótese que esto no es una interrupción; el procesador no salva el contexto y se dedica a hacer otra cosa. En su lugar, el procesador hace una pausa durante un ciclo del bus. El efecto general es hacer que el procesador ejecute con más len-

titud durante una transferencia de DMA. No obstante, para una transferencia de E/S de varias palabras, el DMA es bastante más eficiente que la E/S programada o la dirigida por interrupciones. Cuando el módulo de E/S en cuestión es un sofisticado canal de E/S, el concepto de DMA debe tenerse en cuenta con más razón. Un canal de E/S es un procesador propiamente dicho, con un conjunto especializado de instrucciones, diseñadas para la E/S. En un sistema informático con tales dispositivos el procesador no ejecuta instrucciones de E/S. Dichas instrucciones se almacenan en la memoria principal para ser ejecutadas por el propio canal de E/S. Así pues, el procesador inicia una transferencia de E/S instruyendo al canal de E/S para ejecutar un programa en memoria. El programa especifica el o los dispositivos, la zona o zonas de memoria para almacenamiento, la prioridad y la acción que llevar a cabo bajo ciertas condiciones de error. El canal de E/S sigue estas instrucciones y controla la transferencia de datos, informando de nuevo al procesador al terminar.

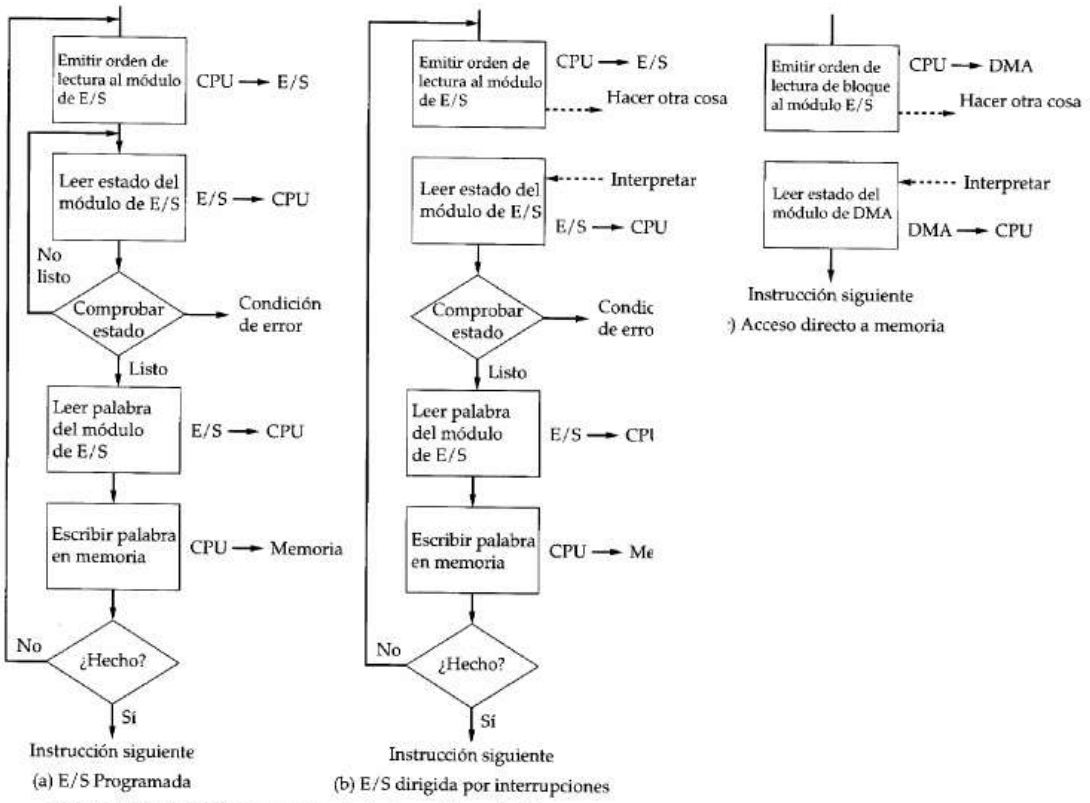


FIGURA 1.21 Tres técnicas para la entrada de un bloque de datos

For what types of operations is DMA useful? Explain your answer.
Answer: DMA is useful for transferring large quantities of data between memory and devices. It eliminates the need for the CPU to be involved in the transfer, allowing the transfer to complete more quickly and the CPU to perform other tasks concurrently.

11 Spool:

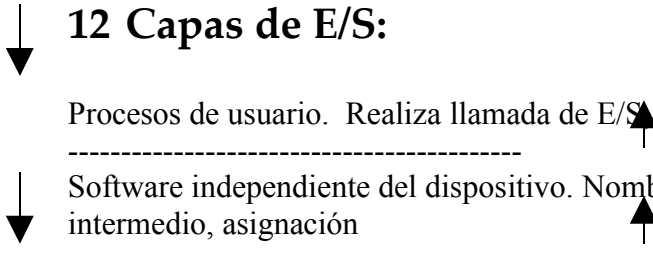
-El uso de spool es una forma de manejar los dispositivos de E/S de uso exclusivo en un sistema multiprogramado. Consideremos un dispositivo de spool típico: una impresora. Aunque desde el punto de vista técnico seria fácil dejar que cualquier proceso de usuario abriera el archivo especial por caracteres de la impresora, podría suceder que un proceso lo abriera y luego pasara horas sin hacer nada. Ningún otro proceso podría imprimir nada.

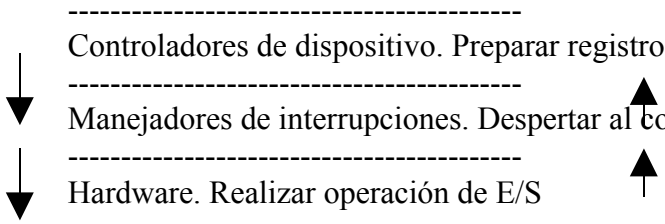
En vez de ello, lo que se hace es crear un proceso especial, llamado genéricamente demonio, y un directorio especial, llamado directorio de spool. Si un proceso quiere imprimir un archivo, primero genera el archivo completo que va a imprimir y lo coloca en el directorio de spool. Corresponde al demonio, que es el único proceso que tiene permiso de usar el archivo especial de la impresora, escribir los archivos en el directorio. Al proteger el archivo especial contra el uso directo por parte de los usuarios, se elimina el problema de que alguien lo mantenga abierto durante un tiempo innecesariamente largo.

El spool no se usa sólo para impresoras; también se usa en otras situaciones. Por ejemplo, es común usar un demonio de red para transferir archivos por una red.

Prefetching is a method of overlapping the I/O of a job with that job's own computation. The idea is simple. After a read operation completes and the job is about to start operating on the data, the input device is instructed to begin the next read immediately. The CPU and input device are then both busy. With luck, by the time the job is ready for the next data item, the input device will have finished reading that data item. The CPU can then begin processing the newly read data, while the input device starts to read the following data.
A similar idea can be used for output. In this case, the job creates data that are put into a buffer until an output device can accept them.
Compare the prefetching scheme with the spooling scheme, where the CPU overlaps the input of one job with the computation and output of other jobs.
Answer: Prefetching is a user-based activity, while spooling is a system-based activity. Spooling is a much more effective way of overlapping I/O and CPU operations.

12 Capas de E/S:





13 Proceso:

Un proceso es un programa en ejecución, la cual debe proceder de manera secuencial. Un proceso además del código del programa, incluye la actividad actual, representada por el valor del *contador de programa*, y el valor de los registros de la CPU. Generalmente también incluye al *stack* del proceso, que contiene datos temporales, y una sección de datos que contiene variables globales.

Un programa, es una entidad pasiva, como un archivo en disco, mientras que un proceso es una entidad activa.

Aunque podría haber dos procesos asociados al mismo programa, de todas maneras se consideran como dos secuencias de ejecución distintas.

13.1.1 CONTROL DE PROCEDIMIENTOS

Una técnica común para controlar las llamadas a procedimientos y los retornos es usar una pila. Este apéndice resume las propiedades básicas de las pilas y analiza su utilización en el control de los procedimientos.

13.1.1.1 Implementación de las pilas

La implementación de la pila exige un cierto número de posiciones empleadas para almacenar los elementos. En la figura 1.27 se ilustra un enfoque típico de implementación. Se reserva en la memoria principal (o en la virtual) un bloque de posiciones contiguas para la pila. La mayor parte del tiempo, el bloque estará parcialmente lleno con los elementos de la pila y el resto permanecerá disponible para su crecimiento. Se necesitan tres direcciones para las operaciones, las cuales se almacenan a menudo en los registros del procesador:

• *Un puntero de pila:* Contiene la dirección de la cima de la pila. Si se añade un elemento a la pila (METER) o se elimina un elemento de la pila (SACAR), este puntero se incrementa o se decrementa para poder tener la dirección de la nueva cima de la pila.

• *Base de la pila:* Contiene la dirección del fondo de la pila en el bloque reservado para la misma. Ésta es la primera posición que se utiliza cuando se añade un elemento a una lista vacía. Si se intenta SACAR cuando la pila está vacía, se informará sobre el error.

• *Límite de la pila:* Contiene la dirección del otro extremo del bloque reservado para la pila. Si se intenta METER cuando la pila está llena, se informará sobre el error.

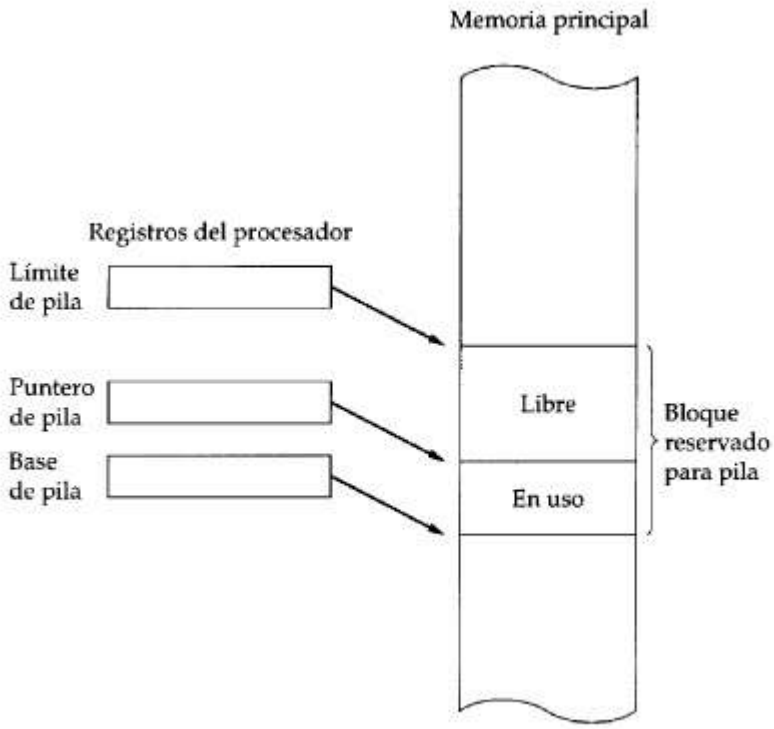


FIGURA 1.27 Organización típica de una pila

13.1.1.2 Llamadas a procedimientos y retornos

Una técnica habitual para gestionar las llamadas a procedimientos y los retornos se basa en el uso de una pila. Cuando el procesador ejecuta una llamada, pone la dirección de retomo en la pila. Cuando ejecuta un retomo, utiliza la dirección que está en la cima de la pila. La figura 1.28 ilustra el uso de la pila para administrar los procedimientos anidados de la figura 1-29.

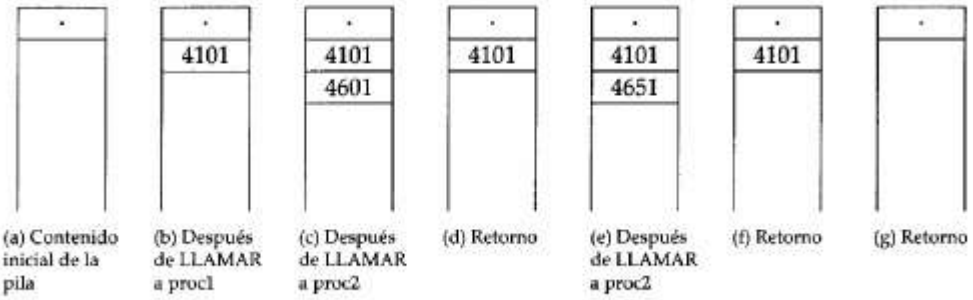


FIGURA 1.28 Empleo de una pila para implementar los procedimientos anidados de la figura 1.29

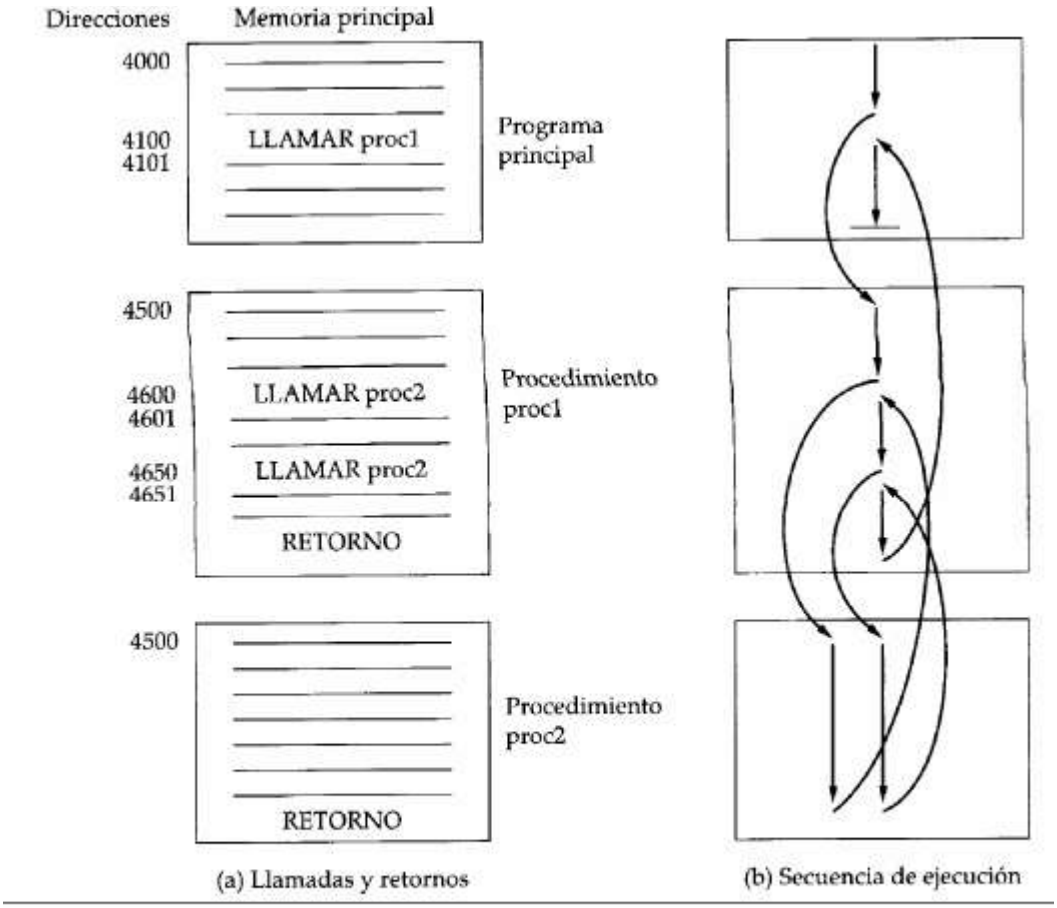


FIGURA 1.29 Procedimientos Anidados

Además de dar la dirección de retomo, a menudo es necesario pasar unos parámetros en las llamadas a procedimientos. Estos podrían pasarse en registros. Otra posibilidad es alma- cenar los parámetros en la memoria justo antes de la instrucción LLAMAR *(CALL).* En tal caso, el retomo debe ser a la posición que sigue a los parámetros. Ambos enfoques tienen sus inconvenientes. Si se utilizan los registros, el programa llamado deberá escribirse de modo que se asegure que los registros se usan correctamente. Almacenar los parámetros en memoria dificulta el intercambio de un número variable de parámetros. Un método más flexible de paso de parámetros es la pila. Cuando el procesador ejecuta una lla- mada, no sólo apila la dirección de retomo, sino también los parámetros que tiene que pasarle al procedimiento llamado. El procedimiento llamado puede acceder a los parámetros en la pila. Al volver, los parámetros de retomo también pueden ponerse en la pila, *bajo* la dirección de retomo. El conjunto completo de parámetros, incluyendo la dirección de retomo, que se almacena como producto de la invocación de un procedimiento, es conocido como marco de pila *(snack frame).*

En la figura 1.30 se ofrece un ejemplo. Este se refiere a un procedimiento P en el que se declaran las variables locales *x*1 y x2 y un procedimiento Q, que puede ser llamado por P y en el que se declaran las variables locales y1 e y2. En la figura, el punto de retorno para cada procedimiento es el primer elemento almacenado en el marco de pila correspondiente. A continuación, se almacena un puntero al comienzo del marco anterior. Esto es necesario si el número o la longitud de los parámetros a almacenar son variables.

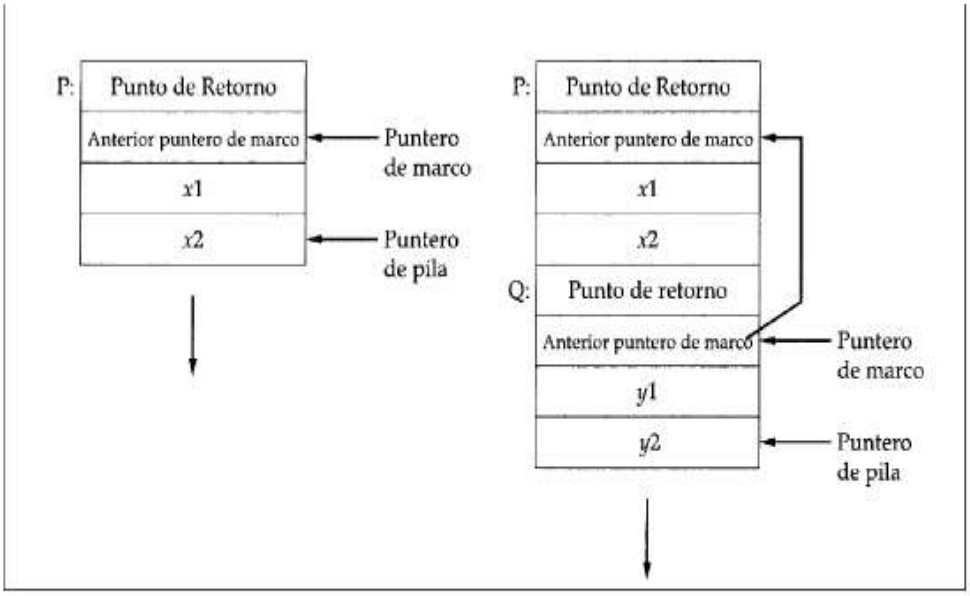


FIGURA 1.30 Crecimiento del marco de pila mediante los procedimientos de muestra P y Q [DEWA90]

13.1.1.3 Procedimientos reentrantes

Un concepto útil, particularmente en un sistema que da soporte a varios usuarios al mismo tiempo, es el de procedimiento reentrante. Un *procedimiento reentrante* es aquel en el que sólo una copia del código del programa puede estar compartida entre varios usuarios durante el mismo periodo. La reentrada tiene dos aspectos clave: El código del programa no puede modificarse a sí mismo y los datos locales para cada usuario se deben almacenar por separado. **Un procedimiento reentrante puede ser interrumpido y llamado por un programa y seguir ejecutando correctamente** al retornar al procedimiento. En un sistema compartido, la reentrada permite un uso más eficiente de la memoria principal: Aunque se almacena una sola copia del código del programa en la memoria principal, más de una aplicación pueden llamar al procedimiento.
Así pues, un procedimiento reentrante debe tener una parte permanente (las instrucciones que constituyen el procedimiento) y una parte temporal (un puntero hacia atrás al procedimiento que llamó, así como memoria para las variables locales utilizadas por el programa). Cada caso particular de ejecución de un procedimiento, que se denomina *activación,* ejecutará el código que está en la parte permanente, pero tendrá su propia copia de las variables locales y de los parámetros. La parte temporal asociada con una activación en particular se conoce como *registro de activación.*
La forma más conveniente de dar soporte a los procedimientos reentrantes es por medio de una pila. Cuando se llama a un procedimiento reentrante, se puede almacenar en la pila el registro de activación de dicho procedimiento. De esta manera, el registro de activación se convierte en parte del marco de la pila que se crea al LLAMAR al procedimiento.

- Informalmente un proceso es un programa en ejecución.

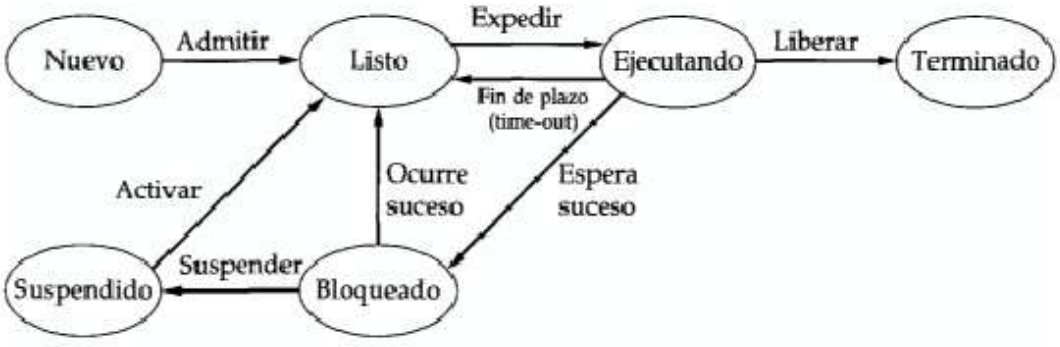
- En cualquier instante, cuando mas una instrucción se estará ejecutando a nombre de un proceso.
- Un proceso esta compuesto por su sección de texto y su sección de datos
- A medida que un proceso se ejecuta, cambia de estado
    - Nuevo – el proceso se esta creando
    - En ejecución – se están ejecutando instrucciones
    - En espera – el proceso esta esperando que ocurra algún suceso (terminación de E/S)
    - Listo – El proceso esta esperando que se le asigne un procesador.
    - Terminado – El proceso termino su ejecución.

- Es importante tener presente que solo un proceso puede estar ejecutándose en cualquier procesador en un instante dado, pero muchos procesos pueden estar listos y esperando.

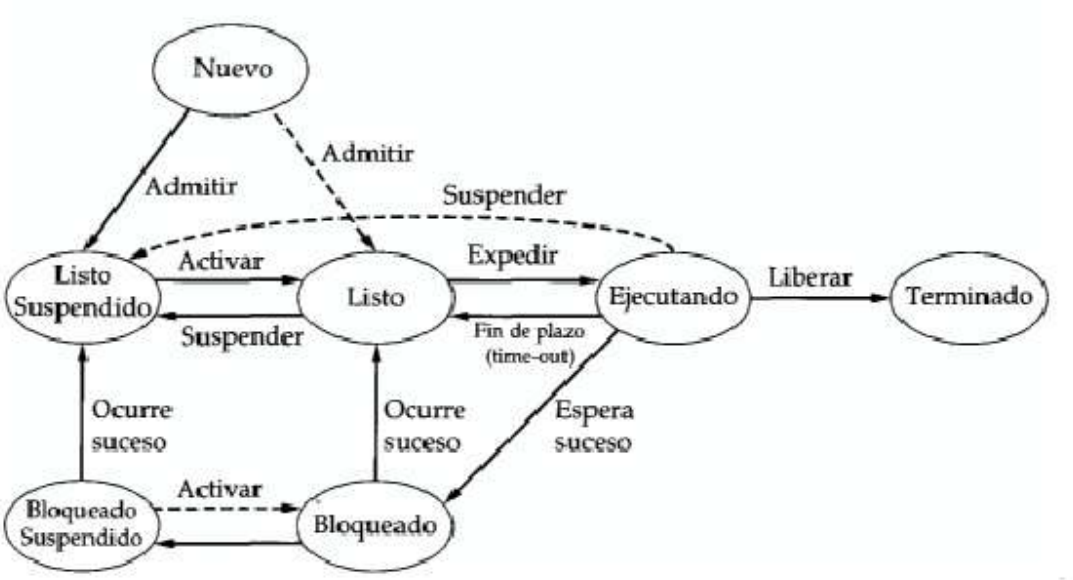
13.1.2 Estados de un proceso (suspendido):

(115 stallings)

-Cuando ninguno de los procesos en memoria principal está en estado listo, el sistema operativo expulsa a disco a uno de los procesos que este bloqueado y lo pasa a una cola de suspendidos. Al emplear intercambio, tal como se acaba de describir, se debe añadir un nuevo estado al modelo de comportamiento de procesos, el estado suspendido. Cuando todos los procesos de memoria principal están en el estado bloqueado, el sistema operativo puede suspender un proceso poniéndolo en el estado suspendido y transferirlo al disco. El espacio que se libera de memoria principal puede utilizarse para traer otro proceso.



(a) Con un estado Suspendido



Listo: El proceso está en memoria principal y listo para ejecución.
Bloqueado: El proceso está en memoria principal esperando un suceso.
Bloqueado y suspendido: El proceso esta en memoria secundaria esperando un suceso.
Listo y Suspendido: El proceso está en memoria secundaria pero está disponible para su ejecución tan pronto como se cargue en la memoria principal.

TABLA 3.6 Razones para la Suspensión de procesos

Intercambio	El sistema operativo necesita liberar suficiente memoria principal para cargar un proceso que está listo para ejecutarse.
Otra razón del SO	El sistema operativo puede suspender un proceso de fondo, de utilidad o cualquier proceso que se sospecha sea el causante de un problema.
Solicitud de un usuario con	Un usuario puede querer suspender a ejecución de un programa fines de depuración o en conexión con el uso de un recurso.
Por tiempo	Un proceso puede ejecutarse periódicamente (por ejemplo, un proceso de contabilidad o de supervisión del sistema) y puede ser suspendido mientras espera el siguiente intervalo de tiempo.
Solicitud del proceso padre	Un proceso padre puede querer suspender a ejecución de un descendiente para examinar o modificar el proceso suspendido o para coordinar la actividad de varios descendientes.

13.1.3 Estructuras de control del sistema operativo:

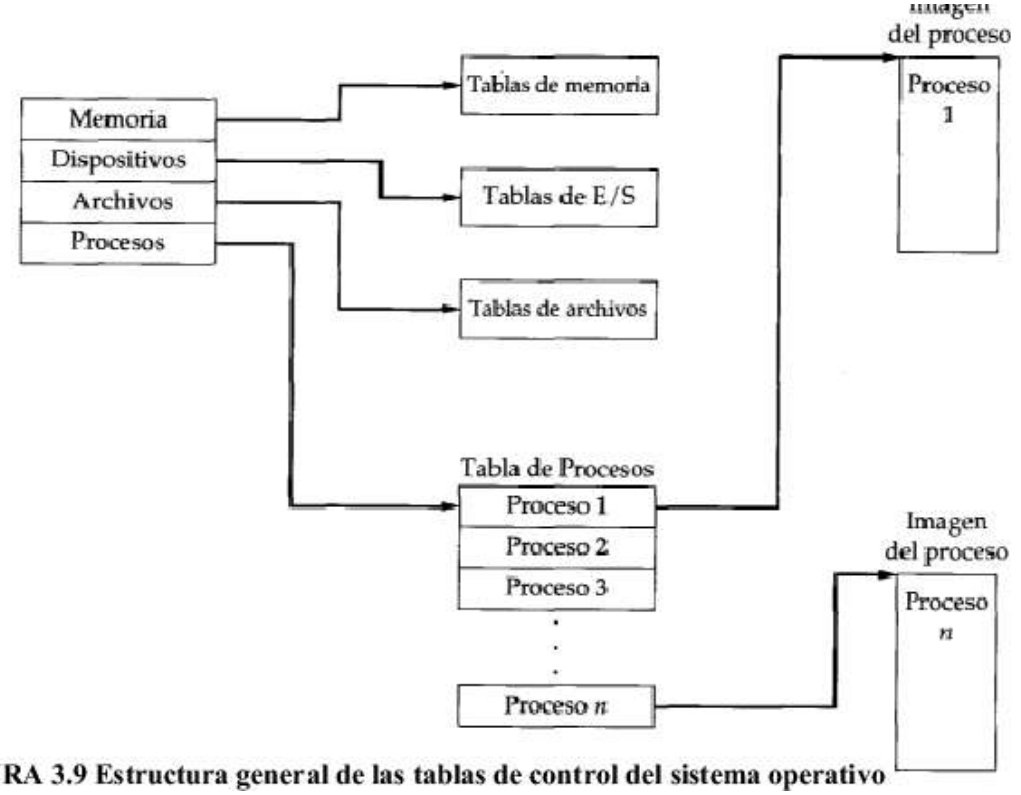


FIGURA 3.9 Estructura general de las tablas de control del sistema operativo

Hay cuatro tipos de tabla que por lo general mantiene el sistema operativo:

-De memoria
- -asignacion de memoria principal a los procesos
- -asignacion de memoria secundaria a los procesos
- -atributos de proteccion de la memoria virtual o principal (que procesos pueden acceder a ciertas regiones compartidas de memoria)
- -informacion de gestion de la memoria virtual

-De E/S
- -Administran los dispositivos y los canales de E/S del sistema informatico.

-De archivos
- -Ofrecen informacion acerca de la existencia de los archivos, su posicion en la memoria secundaria, su estado actual y otros atributos.

-De procesos

13.1.3.1 PCB:

la informaci´on del bloque de control del proceso se puede
agrupar en tres categor´_as distintas:
_ Identi_caci´on del proceso, con los identi_cadores del proceso, del

proceso padre que lo cre´o y del usuario.
_ Informaci´on del estado del procesador, con los registros visibles por el usuario, los registros de control y de estado y los punteros de pila.
_ Informaci´on de control del proceso, con informaci´on de plani_caci ´on y de estado, relaciones con otros procesos, los privilegios, la gesti ´on de la memoria y la propiedad de los recursos y su utilizaci ´on.

- Cada proceso se presenta en el sistema operativo con un bloque de control de proceso (PCB).
- Sus elementos son los siguientes:
    - Estado del proceso
    - Contador de programa
    - Registros de CPU
    - Información de planificación de CPU (prioridad del procesos, punteros a colas)
    - Información de gestión de memoria (valor de los registros base y limite, tabla de paginas, o tabla de segmentos)
    - Información contable (cantidad de tiempo de CPU y tiempo real consumida, limites de tiempo, numero de proceso, y demás).
    - Información de estado de E/S (lista de dispositivos de E/S asignados al proceso, lista de archivos abiertos, etcétera).

-El objetivo de la multiprogramación es tener algún proceso en ejecución en todo momento, a fin de maximizar el aprovechamiento de la CPU. El objetivo del tiempo compartido es conmutar la CPU entre procesos con tal frecuencia que los usuarios puedan interactuar con cada programa durante su ejecución.
-En el caso de un sistema con un solo procesador, nuca habrá mas de un programa en ejecución
-Conforme los procesos ingresan en el sistema, se colocan en una cola de trabajos. Esta cola incluye todos los procesos del sistema.

-Los procesos que están en la memoria principal y están listos y esperando para ejecutarse se mantienen en una lista llamada cola de procesos listos. Cada PCB tiene un campo de puntero que apunta al siguiente proceso de la cola de procesos listos.
Hay otras colas en el sistema. La lista de procesos que esperan un dispositivo de E/S en particular se denomina cola de dispositivo. Cada dispositivo tiene su propia cola.

13.1.3.2 Planificadores:

-Un proceso migra de una cola de planificación a otra durante toda su existencia.
El planificador es el encargado de seleccionar el proceso a migrar.

-En un sistema por lotes, es común que haya más procesos presentados de los que se pueden ejecutar inmediatamente. Estos procesos se colocan en spool en un dispositivo de almacenamiento masivo, y ahí esperan hasta que pueden ejecutarse. El planificador a largo plazo escoge procesos de esta reserva y los carga en la memoria para que se ejecuten. El planificador a corto plazo (planificador de la CPU) escoge entre los procesos que están listos para ejecutarse, y asigna la cpu a una de ellos.
-La distinción primaria entre estos dos planificadores es la frecuencia con que se ejecutan. El planificador a corto plazo debe seleccionar un proceso para la cpu de manera relativamente frecuente. El planificador a largo plazo se ejecuta con una frecuencia mucho menor, controla el grado de multiprogramación (el numero de procesos en memoria).
Es importante que el planificador elija una buena mezcla de procesos limitados por CPU y por E/S. Algunos sistemas de tiempo compartido poseen un planificador a mediano plazo, los cuales en determinadas ocasiones sacan procesos de la memoria a fin de reducir el grado de multiprogramación, mandando a disco un proceso (swaping) y luego lo retorna a memoria.

Describe the differences among short-term, medium-term, and long-term scheduling.
Answer:
Short-term(CPU scheduler)—selects from jobs in memory those jobs that are ready to execute and allocates the CPU to them.
Medium-term—used especially with time-sharing systems as an intermediate scheduling level. A swapping scheme is implemented to remove partially run programs from memory and reinstate them later to continue where they left off.
Long-term (job scheduler)—determines which jobs are brought into memory for processing.
The primary difference is in the frequency of their execution. The short-term must select a new process quite often. Long-term is used much less often since it handles placing jobs in the system and may wait a while for a job to finish before it admits another one.

13.1.3.3 Conmutación de contexto:

El cambio de la CPU a otro proceso requiere guardar el estado del proceso anterior y cargar el estado guardado del nuevo proceso. Esta tarea se denomina conmutación de contexto (context switch). El tiempo de conmutación es exclusivamente gasto extra (overhead), porque el sistema no realiza trabajo útil durante la conmutación. La conmutación de contexto se ha convertido en un cuello de botella tan importante para el desempeño que se están empleando estructuras nuevas (hilos) para evitarlas hasta donde sea posible.

Describe the actions a kernel takes to context switch between processes.
Answer: In general, the operating system must save the state of the currently running process and restore the state of the process scheduled to be run next. Saving the state of a process typically includes the values of all the CPU registers in addition to memory allocation. Context switches must also performmany architecture-specific operations, including flushing data and instruction caches.

13.1.3.4 Hilos:

A falta de una de_nici´on m´as formal, un hilo se puede de_nir como la traza de ejecuci ´on de un proceso. Esto se re_ere a la parte din´amica del proceso, la ejecuci ´on del c´odigo representada por la pila del usuario y la parte del bloque de control del proceso que hace referencia al estado del procesador y del proceso, frente a la parte m´as est´atica, como es la el resto del bloque de control del proceso y el c´odigo mismo. La aparici ´on de los hilos viene justi_cada desde dos pilares b´asicos: facilitar la sincronizaci´on entre procesos y mejorar la e_ciencia en la alternancia de procesos en el procesador.

-Un proceso tiene su propio contador de programa, registro de pila y espacio de direcciones.
-Compartir recursos y acceder a ellos de forma concurrente.
-Un hilo (proceso ligero), es una unidad básica de utilización de la CPU, y consiste en un contador de programa, un juego de registros y un espacio de pila. El hilo comparte con sus hilos pares la sección de código, sección de datos recursos del sistema operativo como archivos abiertos y señales, lo que se denomina colectivamente una tarea. Un proceso tradicional o pesado es igual a una tarea con un solo hilo.

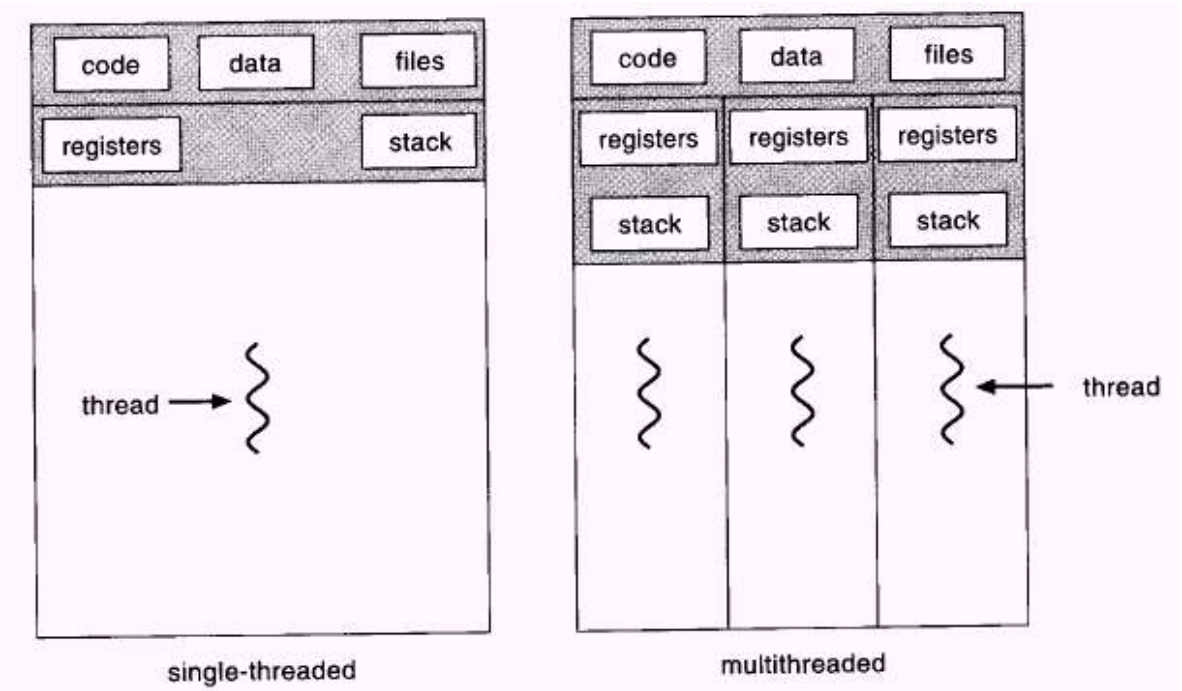


Figure 5.1 Single- and multithreaded processes.

A diferencia de los procesos, los hilos no son independientes entre si y dado que todos pueden acceder a todas las direcciones de la tarea, un hilo puede leer o escribir sobre la pila de otro.

La estructura no ofrece protección entre hilos, sin embargo, esa protección no debería ser necesaria, puesto que el dueño de la tarea con múltiples hilos, es un único usuario.

-Aunque una conmutación de contexto de hilos también requiere un cambio de conjunto de registros, no hay que realizar operaciones relacionadas con la gestión de memoria.

-Algunos sistemas implementan hilos en el nivel de usuario en bibliotecas en el nivel de usuario, no por medio de llamadas al sistema, así que la conmutación de hilos no necesita invocar el sistema operativo ni causar una interrupción para pasar al núcleo. La conmutación entre hilos en el nivel de usuario puede hacerse con independencia del sistema operativo y, por tanto, muy rápidamente. Así bloquear un hilo y conmutar a otro es una solución razonable al problema de cómo un servidor puede manejar muchas solicitudes de forma eficiente. Los hilos en nivel de usuario tienen desventajas. Por ejemplo, si el núcleo es monohilado, cualquier hilo en el nivel de usuario que ejecute una llamada al sistema hará que toda la tarea espere hasta que la llamada regrese.

-Un proceso es útil cundo los trabajos que realizan no tienen relación unos con otros. Cuando múltiples procesos realizan la misma tarea es mejor utilizar hilos.

-Un proceso multihilado emplea menos recursos que varios procesos redundantes, incluida memoria, archivos abiertos y planificación de CPU.

-Un hilo dentro de un proceso se ejecuta secuencialmente y cada hilo tiene su propia pila y contador de programa. Los hilos pueden crear hilos hijos y se pueden bloquear en espera a que termine la ejecución de llamadas al sistema; si un hilo se bloquea, otro puede ejecutarse. Por otro lado a diferencia de los procesos los hilos no son independientes entre si.

-En los hilos en nivel de usuario no interviene el núcleo, y por tanto la conmutación entre ellos es más rápida que entre hilos apoyados por el núcleo. Sin embargo, las llamadas al sistema operativo pueden hacer que todo el proceso espere, porque el núcleo solo planifica procesos (ya que no tiene conocimiento de los hilos), y un proceso que está esperando no recibe tiempo de CPU. Además la planificación podría no ser equitativa.

-En los sistemas con hilos apoyados por el núcleo, la conmutación entre los hilos consume más tiempo porque el núcleo debe realizarla (a través de una interrupción)

-Otra forma en la que los hilos aportan eficiencia es en la comunicación entre diferentes programas en ejecución. En la mayoría de los sistemas operativos, la comunicación entre procesos independientes requiere la intervención del núcleo para ofrecer protección y para proporcionar los mecanismos necesarios para la comunicación. Sin embargo, puesto que los hilos de una misma tarea comparten memoria y archivos, pueden comunicarse entre si sin invocar al núcleo.

-Los hilos a nivel de núcleo no tienen porque estar asociados a un proceso pero cada hilo de usuario pertenece a un proceso.

- ¿Que recursos son usados cuando un hilo es creado? ¿Como difieren de aquellos usados cuando un proceso es creado?

  Dado que un hilo es mas pequeño que un proceso, la creación de hilos típicamente usa menos recursos que la creación de un proceso. Crear un proceso requiere reservar un PCB, el cual es una estructura de datos grande. El PCB incluye mapeo de memoria, lista de archivos abiertos, y variables de entorno. Reservar y administrar el mapa de memoria es típicamente la actividad que consume mas tiempo. Crear un hilo de usuario o de núcleo implica reservar una pequeña estructura de datos, para mantener el conjunto de registros, la pila, y su prioridad.

-Entre los elementos que son distintos para cada hilo están el contador de programa, los registros y el estado. El contador de programa se necesita para que los hilos al igual que los procesos puedan suspenderse y reanudarse. Los registros se necesitan porque cuando los hilos se suspenden sus registros deben guardarse.

-La conmutacion entre hilos es mucho mas rapida cuando se hace en espacio de ususario;Por otro lado cuando la administracion se realiza totalmente en espacio de usuario y uno se bloquea, el kernel bloquea todo el proceso ya que no tiene idea de que existen otros hilos.

What are two differences between user-level threads and kernel-level threads? Underwhat circumstances is one type better than the other?
Answer: (1) User-level threads are unknown by the kernel, whereas the kernel is aware of kernel threads. (2) User threads are scheduled by the thread library and the kernel schedules kernel threads. (3) Kernel threads need not be associatedwith a process whereas every user thread belongs to a process

Describe the actions taken by a kernel to context switch between kernel-level threads.
Answer: Context switching between kernel threads typically requires saving the value of the CPU registers from the thread being switched out and restoring the CPU registers of the new thread being scheduled.

Describe the actions taken by a thread library to context switch between user-level threads.
Answer: Context switching between user threads is quite similar to switching between kernel threads, although it is dependent on the threads library and how it maps user threads to kernel threads. In general, context switching between user threads involves taking a user thread of its LWP and replacing it with another thread. This act typically involves saving and restoring the state of the registers.

What resources are used when a thread is created? How do they differ from those used when a process is created?
Answer: Because a thread is smaller than a process, thread creation typically uses fewer resources than process creation. Creating a process requires allocating a process control

block (PCB), a rather large data structure. The PCB includes a memory map, list of open files, and environment variables. Allocating and managing the memory map is typically the most time-consuming activity. Creating either a user or kernel thread involves allocating a small data structure to hold a register set, stack, and priority.

Describa las acciones del kernel de un sistema operativo cuando ocurre un cambio de contexto :

1. entre hilos
2. entre procesos

Para el caso de un cambio de contexto entre hilos de un mismo proceso, el sistema operativo debe salvar los registros, el contador de programa y el registro de pila del hilo en el PCB. Sin embargo no se requiere ninguna otra gestión relativa al espacio de memoria u otros recursos, ya que el contiene a los hilos. De acuerdo a lo expuesto, para un proceso activo el planificador podría priorizar el despacho de hebras del mismo en el procesador o cpu corriente.
Si los hilos pertenecen a tareas o procesos diferentes obviamente se hace necesario conmutar el contexto de acuerdo a la técnica habitual.

Para el caso de cambio de contexto entre procesos diferentes, el sistema operativo debe realizar las mismas tareas que con los hilos, pero además debe movilizar información relativa a los recursos asignados al proceso, como ser bloques de memoria o disco asignados y procesos descendientes del proceso actual.

[sta]
Los beneficios clave de los hilos se derivan de las implicaciones del rendimiento: Se tarda mucho menos tiempo en crear un nuevo hilo en un proceso existente que en crear una nueva tarea, menos tiempo para terminar un hilo y menos tiempo para cambiar entre dos hilos de un mismo proceso. Por tanto, si hay una aplicación o una función que pueda implementarse como un conjunto de unidades de ejecución relacionadas, es más eficiente hacerlo con una colección de hilos que con una colección de tareas separadas.[8] Algunos estudios llevados a cabo por los desarrolladores de Mach demuestran que la aceleración en la creación de procesos, comparada con la de Las implementaciones de UNIX que no utilizan hilos, está en un factor de 10 [ITEVA87].

La planificación y la expedición se llevan a cabo con los hilos; por tanto, la mayor parte de la información de estado relacionada con la ejecución se mantiene en estructuras de datos al nivel de los hilos. Sin embargo, hay varias acciones que afectan a todos lo hilos de una tarea y que el sistema operativo debe gestionar al nivel de las tareas. La suspensión implica la descarga del espacio de direcciones fuera de la memoria principal. Puesto que todos los hilos de una tarea comparten el mismo espacio de direcciones, todos deben entrar en el estado Suspendido al mismo tiempo. De manera similar, la terminación de una tarea supone terminar con todos los hilos dentro de dicha tarea.

[tan]
La conmutación de hilos es mucho más rápida cuando la administración de hilos se efectúa en el espacio de usuario que cuando se necesita una llamada al kernel. Este hecho es un argumento convincente para realizar la administración de hilos en el espacio de usuario. Por otro lado, cuando los hilos se manejan totalmente en el espacio de usuario y uno se bloquea (p. ej., esperando E/S o

que se maneje una falla de página), el kernel bloquea todo el proceso, ya que no tiene idea de que existen otros hilos. Este hecho es un argumento importante en favor de realizar la administración de hilos en el kernel. La consecuencia es que se usan ambos sistemas; además, se han propuesto diversos esquemas híbridos (Anderson et al., 1992).

[pdf hilos]
En un entorno multihilo un proceso se define como la unidad de
proteccion y unidad de asignacion de recursos con los siguientes elementos
asociados con ellos:
- Un espacio de direcciones virtuales con la imagen del proceso.
- Acceso protegido a los procesadores, a otros procesos, archivos
y dispositivos de E/S.
Un proceso puede tener uno o mas hilos, cada uno con:
- El estado de ejecucion del hilo.
- El contexto del procesador que se salva cuando el hilo no esta en
ejecucion.
- Una pila de ejecucion de usuario (y otra para el nucleo cuando
sea preciso).
_ Almacenamiento estatico para las variables locales.
_ Acceso a la memoria y a los recursos del proceso, que son compartidos
con todos los otros hilos del proceso.
Asi pues, todos los hilos de un proceso comparten el estado y los
recursos del proceso, residen en el mismo espacio de direcciones y
tienen acceso a los mismos datos. Por todo esto, es facil ver que la utilizacion de hilos produce una mejora en las posibilidades de sincronizacion de procesos, ademas de una mejora en el rendimiento. Las ventajas que se derivan de la utilizacion de los hilos son las siguientes:

1. *Respuesta*: Una aplicacion multihilo puede continuar ejecutandose
incluso si una parte de ella esta bloqueada o esta realizando una
tarea larga, incrementando de este modo el nivel de respuesta
frente al usuario.
2. *Comparticion de recursos*: Los hilos por defecto comparten la memoria
y otros recursos del proceso al que pertenecen por lo que
no hay que utilizar ningun mecanismo especial de comparticion
entre procesos. Esto permite que la aplicacion pueda tener distintos
hilos de ejecucion con la misma memoria. Ademas los hilos
aumentan la eficiencia de al comunicacin entre programas
en ejecucion ya que pueden comunicarse sin la intervencin del
nucleo del sistema operativo.
3. *Economia*: Reservar memoria y recursos en la creacion de los procesos
es una tarea costosa. Como los hilos comparten los recursos
del proceso al que pertenecen, resulta mas economico crear
hilos nuevos y alternarlos en el procesador que hacer lo mismo
con procesos. Aunque no existen medidas empiricas definitivas,
en Solaris crear un proceso es unas 30 veces mas lento que crear
un hilo, y realizar un cambio de contexto entre procesos es 5 veces
mas lento que hacerlo entre hilos.
4. *Utilizacion de arquitecturas multiprocesador*: Los procesos multihilo
pueden aprovechar facilmente una arquitectura multiprocesador
ya que cada hilo podria ejecutarse en un procesador distinto.

Las ventajas de utilizar HU en lugar de HN son las siguientes:
1. El intercambio entre dos hilos no necesita los privilegios del n´ ucleo, lo que evita la sobrecarga de los cambios de modo de ejecuci ´on.
2. Se puede realizar una plani_caci´on a medida de la aplicaci ´on sin afectar a la plani_caci´on subyacente del sistema operativo.
3. Los HU se pueden ejecutar sobre cualquier sistema operativo.
Tambi´en existen desventajas:
1. Como la mayor´_a de las llamadas al sistema producen bloqueo en el proceso que las invoca, la llamada al sistema de un hilo puede bloquear a todos los dem´as hilos del proceso (Figura 9 e).
2. No puede aprovechar las ventajas de un entorno multiprocesador, ya que el proceso est´a asignado a un ´ unico procesador.

Hilos a Nivel de N´ ucleo

El n´ ucleo mantiene la informaci´on de contexto del proceso como un todo y la de cada hilo dentro del proceso. El n´ ucleo realiza la plani_caci´on en funci´on de los hilos. Esto soluciona los principales inconvenientes de la implementaci´on HU. Se pueden plani_car simult´aneamente m´ ultiples hilos del mismo proceso en un entorno multiprocesador. Adem´as, si se bloquea un hilo del proceso los dem´as siguen pudiendo ejecutarse. Otra ventaja es que las mismas funciones del n´ ucleo pueden ser multihilo.
La principal desventaja es que el paso del control de un hilo a otro dentro del mismo proceso necesita un cambio de modo de ejecuci ´on a modo n´ ucleo. Eso sugiere que aunque el enfoque HN sea m´as r´apido que el basado en monohilo, todav´_a resulte m´as lento que el HU.

13.1.4 Llamada al sistema fork:

La creación de procesos en UNTX se hace por medio de la llamada *fork* al núcleo del sis-tema. Cuando un proceso emite una petición de *fork,* el sistema operativo realiza las siguientes operaciones ACH86J:

1. Asigna una entrada en la tabla de procesos para el nuevo proceso.
2. Asigna un ID único de proceso al proceso hijo.
3. Hace una copia de La imagen del proceso padre, a excepción de la memoria compartida.
4. Incrementa los contadores de los archivos que son propiedad del padre, para reflejar el hecho que hay un nuevo proceso ahora que también es propietario de esos archivos.
5. Pone al proceso hijo en el estado Listo para Ejecutar.
6. Devuelve al proceso padre el número de ID del hijo y devuelve un valor cero al proceso hijo.

Todo este trabajo se acomete en el modo núcleo en el proceso padre. Cuando el núcleo haya terminado con estas funciones, podrá realizar alguna de las siguientes como parte de la rutina distribuidora:

1. Permanecer en el proceso padre. El control vuelve al modo de usuario en el punto de la llamada fork del padre.
2. Transferir eL control al proceso hijo. El proceso hijo comienza ejecutando en el mismo punto del código que el padre, es decir, en el punto de retomo de la llamada fork.
3. Transferir el control a otro proceso. Tanto el padre como el hijo se dejan en el estado Listo para Ejecutar.

Puede que sea difícil hacer ver este método de creación de procesos, porque tanto el padre como el hijo están ejecutando el mismo trozo de código. La diferencia es la siguiente:
Cuando se retorna del fork, se pregunta por el parámetro de retomo. Si el valor es cero, entonces se está en el proceso hijo y se puede ejecutar una bifurcación hacia el programa de usuario apropiado para continuar la ejecución. Si el valor es diferente de cero, entonces se está en el proceso padre y puede continuar la línea principal de ejecución.

14 Planificación de la CPU:

-La cola de procesos listos puede implementarse como cola FIFO, cola por prioridad, árbol o simplemente como lista enlazada no ordenada.
-Los registros de las colas generalmente son los PCB de los procesos.

14.1.1 Planificación expropiativa:

-Las decisiones de planificacion de la CPU se toman en las cuatro situaciones siguientes:

i) Cuando un proceso pasa del estado en ejecución al estado en espero (por ejemplo, solicitud de E/S o invocacion de una espera hasta que termine uno de los procesos hijos).

ii) Cuando un proceso pasa del estado en ejecución al estado listo (por ejemplo cuando ocurre una interrupción, se crea un nuevo proceso, otro proceso termino su E/S, terminacion del quantum)

iii) Cuando un proceso pasa del estado en espera al estado listo (por ejemplo, terminacion de E/S)

iv) Cuando un proceso termina

-Con planificación no expropiativa, una vez que la CPU se ha asignado a un proceso, éste la conserva hasta que la cede ya sea porque termino o porque paso al estado en espera.

14.1.1.1 Despachador:

Este modulo es el que cede el control de la CPU al proceso seleccionado por el planificador a corto plazo.

Esto implica:

- Cambiar de contexto
- Cambiar a modo de usuario
- Saltar al punto apropiado del programa de usuario para reiniciar ese programa.

Algoritmos de planificacion:

14.1.2 Planificacion por orden de llegada (FCFS):

-Con este esquema el proceso que primero solicita la cpu la recibe primero; Fácil de implementar con una cola FIFO.

-Posee tiempo de espera muy largo.

-No es expropiativo
-No es adecuado para sistemas de tiempo compartido (no podemos permitir que un proceso ocupe la CPU durante un periodo prolongado).

-Puede hace esperar a procesos cortos que terminen los mas largos.

-Es no expropiativo

14.1.3 Planificación de “primero el trabajo más corto” (SJF):

-Cuando la CPU esta disponible se asigna el proceso con su siuiente rafaga mas corta, los empates se resuelven con FCFS.

-La planificación se efectúa examinando la siguiente ráfaga de CPU de un proceso, no su longitud total.

-Es óptimo en cuanto que da el tiempo de espera promedio mínimo para un conjunto dado de procesos.

-La desventaja es que es dificil conocer la duracion de la proxima rafaga de CPU.

-No se puede implementar en el nivel de planificación a corto plazo. Una alternativa es tratar de aproximimar la siguiente rafaga como un promedio exponencial de las duraciones medidas de ráfagas de CPU previas.

$$tn+1=a\ tn + (1 – a)tn.$$

-Puede causar inanición.

-Puede ser o no expropiativo.

14.1.4 Planificación por prioridad:

-Se asigna una prioridad a cada proceso y se asigna la CPU al proceso con la prioridad mas alta. El empate se resuelve por FCFS.

-El algoritmo SJF es un caso especial del algoritmo de planificación por prioridad general con prioridad igual al reciproco de la duración predicha.

-Suponemos que números bajos representan una prioridad alta.

-Puede ser expropiativo como no expropiativo

-El problema de estos algoritmos por prioridad es el bloqueo indefinido o inanición (starvation); Una solución a este problema es el envejecimiento, que consiste en aumentar la prioridad de los procesos que hace mucho tiempo que esperan en el sistema.

14.1.5 Planificación por turno circular (round robin):

-Diseñado especialmente para los sistemas de tiempo compartido.

-Similar a FCFS pero con expropiacion para conmutar entre procesos.

-Se define una unidad pequeña de tiempo llamado cuanto de tiempo o porción de tiempo; La cola de procesos listos se trata como una cola circular; se recorre la cola y se asigna a cada proceso un tiempo igual al cuanto.

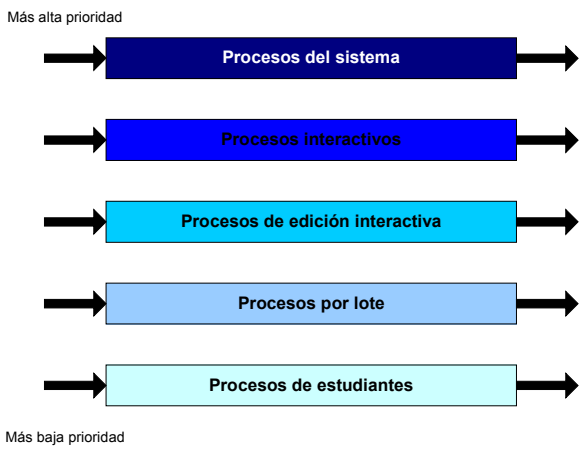
-El tiempo de espera promedio suele ser muy grande.

-Cuando el tiempo del cuanto es muy grande el algoritmo tiende a FCFS, si es muy pequeño la política se llama compartir el procesador, cada uno de los n procesos es como si tuviesen un procesador que ejecuta a 1/n del procesador original.

-Es importante que el cuanto sea grande en relación a lo que tarda la conmutación de contexto.

14.1.6 Planificación con colas de múltiples niveles:

-Útil cuando se puede clasificar a los procesos en grupos (primer plano, interactivos) y (segundo plano, por lotes). Estos dos tipos de procesos tienen diferentes necesidades en cuanto al tiempo de respuesta, así que podrían tener también diferentes necesidades de planificación. Además de poder tener mayor prioridad los de primer plano que los de segundo plano.



-Un algoritmo de planificación por colas de múltiples niveles divide la cola de procesos listos en varias colas distintas. Cada cola tiene su propio algoritmo de planificación.

-Ademas debe haber planificacion entre las colas, lo cual se implementa generalmente como una planificacion expropiativa de prioridades fijas (por ejemplo la cola de primer plano podria tener prioridad absoluta sobre la cola de segundo plano). Por ejemplo que cada cola tenga prioridad absoluta sobre las colas de más baja prioridad; y **ningun proceso de la cola de baja prioridad se pueda ejecutar si no estan vacias las de mayor prioridad**; Otra posibilidad es **dividir el tiempo entre las colas**.

14.1.7 Planificación con colas de múltiples niveles y realimentación:

-La planificacion con colas de multiples niveles y retroalimentacion a **diferencia** de la anterior esquema **permite aun proceso pasar de una cola a otra**.

-Se puede pasar a un proceso de una cola a otra segun sus requerimientos de CPU y E/S, evitando con esta forma de envejecimiento la inanición.

-En general, un planificador de colas multinivel con retroalimentación está definido por los parámetros siguientes:

- El número de colas
- El algoritmo de planificación para cada cola
- El método empleado para determinar cuándo se debe promover un proceso a una cola de mayor prioridad
- El método empleado para determinar cuando se debe degradar a un proceso a una cola de menor prioridad.
- El método empleado para determinar en cual cola ingresará un proceso cuando necesite servicio.

14.1.8 Planificación de múltiples procesadores:

-El problema de planificar se torna mas complejo cuando hay mas procesadores.
-Si se cuenta con varios procesadores identicos puede compartirse la carga.

-Utilizamos una cola común para todos los procesos listos. Todos los procesos ingresan en una cola y se les asigna cualquier procesador que este disponible.

-Hay dos estrategias

- Cada procesador se auto planifica inspeccionando la cola.
- Nombrando un procesador como planificador para los demas, esto evita que dos procesadores quieran despachar el mismo proceso, cosa que puede ocurrir en la estrategia anterior. Esto da lugar a una estrategia Maestro-Esclavo.

Describe the differences between symmetric and asymmetric multiprocessing. What are three advantages and one disadvantage of multiprocessor systems?
Answer: Symmetric multiprocessing treats all processors as equals, and I/O can be processed on any CPU. Asymmetric multiprocessing has one master CPU and the remainder CPUs are slaves. The master distributes tasks among the slaves, and I/O is usually done by the master only. Multiprocessors can save money by not duplicating power supplies, housings, and peripherals. They can execute programs more quickly and can have increased reliability. They are also more complex in both hardware and software than uniprocessor systems.

14.1.9 Planificación en tiempo real:

- Se analizan los sistemas de tiempo real blando ya que **los duros no pueden utilizarse en sistemas con memoria virtual o almacenamiento secundario, ya que causan una variacion impredecible.**
- El sistema debe contar con planificación por prioridad.
- Los procesos de tiempo real deben tener prioridad mas alta.
- La prioridad de los procesos de tiempo real **no debera degradarse** con el tiempo; Aunque los demas procesos si puedan degradarse.
- La latencia de despacho debe ser pequeña.

14.1.9.1 What is the main difficulty that a programmer must overcome in writing an operating

system for a real-time environment?

Answer: The main difficulty is keeping the operating system within the fixed time constraints of a real-time system. If the system does not complete a task in a certain time frame, it may cause a breakdown of the entire system it is running. Therefore when writing an operating system for a real-time system, the writer must be sure that his schedulingschemes don't allow response time to exceed the time constraint.

MS-DOS provided no means of concurrent processing. Discuss three major complications that concurrent processing adds to an operating system.
Answer:
A method of time sharing must be implemented to allow each of several processes to have access to the system. This method involves the preemption of processes that do not voluntarily give up the CPU (by using a system call, for instance) and the kernel being reentrant (somore than one process may be executing kernel code concurrently).
Processes and system resources must have protections and must be protected from each other. Any given process must be limited in the amount of memory it can use and the operations it can perform on devices like disks.
Caremust be taken in the kernel to prevent deadlocks between processes, so processes aren't waiting for each other's allocated resources.

15 Concurrencia:

(tanenbaum)

Necesitamos que se cumplan cuatro condiciones para tener una buena solución:

1. Dos procesos nunca pueden estar simultáneamente dentro de sus regiones criticas.
2. No pueden suponerse nada acerca de las velocidades o el número de las CPU
3. Ningún proceso que se ejecute fuera de su región crítica puede bloquear a otros procesos
4. Ningún proceso deberá tener que esperar indefinidamente para entrar en su región critica.

15.1.1 Sincronización:

Un proceso cooperativo puede afectar o ser afectado por los demás. Podrían compartir un espacio de direcciones lógicas (código y datos), o solo datos a través de archivos. En el primer caso, es a través de hilos, lo que puede dar pie a inconsistencias.

15.1.1.1.1 Ejemplo de concurrencia

```
FACTORIAL (N: Integer)
BEGIN
        COBEGIN
                a = Semifact (N, ½)
                b = Semifact (N/(2 - 1), 1)
        COEND
        fact = a * b
END
```

Multiplico por separado 1… n/2 y n/2 … n

Todo lo que está entre COBEGIN y COEND se puede ejecutar de forma simultánea.

15.1.1.1.2 Ejemplo de no concurrencia

```
J = 10
COBEGIN
            print j
            j = 1000
COEND
print j
```

La primera impresión, no se si es 10 o 1000, todo depende de que instrucción se ejecute primero.

La segunda impresión siempre es 1000, porque hasta que no se ejecute todo lo que está entre el COBEGIN y COEND no se sigue.

15.1.2 6.1 Antecedentes

Cuando compartimos a través de un buffer, hay que controlar si se vacía o llena. Podemos añadir un contador que indica la cantidad de elementos del buffer.

El problema es que como los procesos que leen y escriben son concurrentes, pueden aparecer problemas al actualizar el valor de esa variable, dependiendo de cómo se ejecuten las instrucciones de máquina. Debemos lograr que solo un proceso a la vez trabaje con el contador.

15.1.2.1.1 Ejemplo:

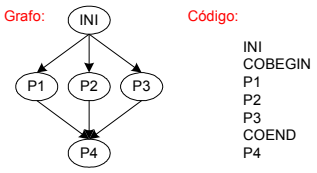
```
1 Reg1 = cont          4 Reg2 = cont
2 Reg1 = Reg1 + 1      5 Reg2 = Reg2 - 1
3 cont = Reg1          6 cont = Reg2
```

La variable contador puede guardar distintos valores. Si el contador al comienzo es 6, y ejecuto las instrucciones:

- 1 – 4 – 2 – 5 – 3 – 6: El contador queda en 5
- 1 – 2 – 3 – 4 – 5 – 6: El contador queda en 6

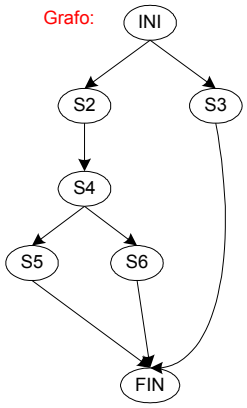
15.1.3 Grafo de precedencia

Es un grafo acíclico, dirigido, cuyos nodos corresponden a las sentencias, y las flechas a la precedencia.



15.1.3.1.1 Ejemplo:

Grafo:



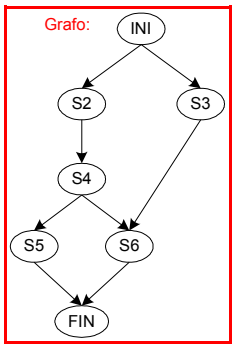
Código 1:

```
INI
COBEGIN
                P
                S3
COEND
FIN

P:
BEGIN
                S2
                S4
                COBEGIN
                                S5
                                S6
                COEND
END
```

Código 2:

```
INI
COBEGIN
                BEGIN
                                S2
                                S4
                                COBEGIN
                                                S5
                                                S6
                                COEND
                END
                S3
COEND
FIN
```

<u>15.1.3.1.2 Ejemplo:</u>

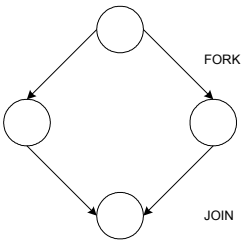


En este caso no hay forma de representar el grafo con COBEGIN y COEND, sin incorporar otras primitivas.

15.1.3.2 FORK – JOIN

```
----------
----------
----------
FORK ET1
----------
----------
----------
ET1
----------
----------
----------
```

Sintaxis: FORK etiqueta JOIN m, etiqueta



Es como un GOTO, donde un proceso se ejecuta siguiente, y el otro después de la etiqueta.

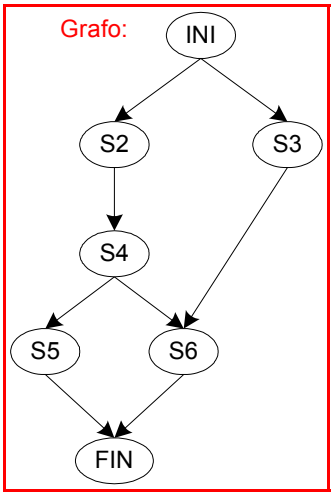
En el caso del JOIN, decremento m en uno, y si m = 0, salto a la etiqueta. Estas dos instrucciones, se ejecutan como instrucción única, de corrido.

Si hay n procesos concurrentes, el join mata uno a uno, y recién el último, pasa a ejecutar el proceso siguiente.

Ejemplo: El caso que no se podía solucionar con COBEGIN Y COEND

```
S1
cont1 = 2
cont2 = 2
FORK L1
```

```
S2
S4
FORK L2
S5
GOTO L7

L1 :
S3

L2:
JOIN cont1, L6
QUIT

L6:
S6

L7:
JOIN cont2, L8
QUIT

L8:
S7
```



15.1.3.3 FORK – WAIT

```
main ()
{
     int pid_hijo;
     pid_hijo = FORK();
     if (pid_hijo < 0)
               SIGNAL_ERROR
     else
     {
               if (pid_hijo == 0)
                     // Soy el hijo
               else
                     // Soy el padre
     }
}
```

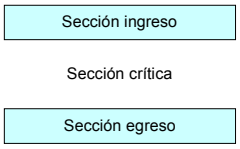
15.1.4 Problema de la exclusion mutua:

```
program      mutualexclusion;
const n = . . . ; (*number of processes*);
procedure P(i: integer);
begin
   repeat
      entercritical (R);
      < critical section >;
      exitcritical (R);
      < remainder >
   forever
end;
begin (* main program *)
   parbegin
      P(1);
      P(2);
      . . .
      P(n)
   parend
end.
```

Figure 5.1 Mutual Exclusion

15.1.4.1 El problema de la sección crítica

Consideramos el sistema con n procesos $\{P_0, \ldots, P_n\}$, donde cada cual tiene un segmento de código, llamado *sección crítica*, en el que el proceso podría estar modificando variables comunes, etc.



La idea es que mientras un proceso se está ejecutando en sección crítica, ningún otro puede estar en otra sección crítica; solo ejecutando otras secciones.

Así la ejecución de secciones críticas, es *mutuamente exclusiva*. Cada proceso debe solicitar permiso para ingresar en su sección crítica y debe indicar cuando salió de ella.

Hay cuatro requisitos:

- **Mutua exclusión:** Si el proceso P_i se ejecuta en su sección crítica, ningún otro proceso puede estarse ejecutando en su sección crítica.
- **Espera limitada:** Hay un límite, para el número de veces que se permite a otros procesos entrar en sus secciones críticas, luego de que un proceso pidió autorización de ingreso. No se permite la *posposición indefinida*.

- **Progreso:** Se deben cumplir las siguientes condiciones:
    - **No alternación:** No puedo obligar a que se ejecuten, primero uno, luego el otro, vuelvo al primero, etc.
    - **Sin puntos muertos (Deadlock):** A espera a B, y B espera a A, y los dos están bloqueados simultáneamente.

15.1.4.2 Soluciones por software:

Algoritmo de Dekker:

```
var flag: array [0 .. 1] of boolean;
        turn: 0 .. 1;
procedure P0;
begin
    repeat
        flag [0] := true;
        while flag [1] do if turn = 1 then
                        begin
                            flag [0] := false;
                                while turn = 1 do { nothing };
                            flag [0] := true
                        end;
        < critical section >;
        turn := 1;
        flag [0] := false;
        < remainder >
    forever
end;
procedure P1;
begin
    repeat
        flag [1] := true;
        while flag [0] do if turn = 0 then
                        begin
                            flag [1] := false;
                                while turn = 0 do { nothing };
                            flag [1] := true
                        end;
        < critical section >;
        turn := 0;
        flag [1] := false;
        < remainder >
    forever
end;
begin
    flag [0] := false;
    flag [1] := false;
    turn := 1;
    parbegin
        P0; P1
    parend
end.
```

Figure 5.3 Dekker's Algorithm

Algoritmo de Petterson

15.1.4.3 soluciones por hardware:

Inhabilitación de interrupciones

En una máquina monoprocesador, la ejecución de procesos concurrentes no puede superponerse; los procesos solo pueden intercalarse. Es más, un proceso continuará ejecutando hasta que solicite un servicio del sistema operativo o hasta que sea interrumpido. Por tanto, para garantizar la exclusión mutua, es suficiente con impedir que un proceso sea interrumpido. Esta capacidad puede ofrecerse en forma de primitivas definidas por el núcleo del sistema para habilitar o inhabilitar las interrupciones. Un proceso puede hacer cumplir la exclusión mutua del siguiente modo (compárese con la figura 4.2):

```
repeat
   <inhabilitar interrupciones>;
   <sección critica>;
   <habilitar interrupciones>;
   <resto>
forever.
```

Puesto que la sección critica no puede ser interrumpida, la exclusión mutua está garantizada. Sin embargo, el precio de esta solución es alto. La eficiencia de la ejecución puede verse notablemente degradada debido a que se limita la capacidad del procesador para intercalar programas. Un segundo problema es que esta técnica no funciona en arquitecturas de multiprocesador. Cuando el sistema tenga más de un procesador, es posible (y habitual) que haya más de un proceso ejecutando al mismo tiempo. En este caso, inhabilitar las interrupciones no garantiza la exclusión mutua.

Instrucciones especiales de la máquina En configuraciones de multiprocesador, varios procesadores comparten el acceso a una memoria principal común. En este caso, no hay una relación maestro/esclavo, sino que los procesadores funcionan independientemente en una relación de igualdad. No hay un mecanismo de interrupciones entre los procesadores en el que se pueda basar la exclusión mutua. A nivel del hardware, como se ha mencionado, los accesos a posiciones de memoria excluyen cualquier otro acceso a la misma posición. Con esta base, los diseñadores han propuesto varias instrucciones de máquina que realizan dos acciones atómicamente, tales como leer y escribir o leer y examinar, sobre una misma posición de memoria en un único ciclo de lectura de instrucción. Puesto que estas acciones se realizan en un único ciclo de instrucción, no están sujetas a injerencias por parte de otras instrucciones. En esta sección se consideran dos de las instrucciones implementadas más habitualmente. Se describen otras en [RAYN86] y [STON90]. *Instrucción Comparar y Fijar* La instrucción Comparar y Fijar (TS, *Test and Set)* puede definirse de la siguiente forma:

```
function TS (var i: entero): booleano;
begin
if i = 0 then
        begin i := I;
        TS := cierto;
         end
else
        TS := falso
 end.
```

La instrucción examina el valor de su argumento, *i.* Si el valor es 0, lo cambia por 1 y de-vuelve cierto. En otro caso, el valor no se modifica y se devuelve falso. La función Comparar y Fijar se ejecuta inmediatamente en su totalidad, es decir, no está sujeta a interrupciones. La figura 4.8a muestra un protocolo de exclusión mutua basado en el uso de esta instrucción. Se da un valor 0 inicial a una variable compartida *cerrojo*. El único proceso que puede entrar en su sección crítica es el que encuentre *cerrojo* igual a 0. Todos los demás procesos que intenten entrar pasan a un **modo de espera activa.** Cuando un proceso abandona su sección critica, vuelve a poner *cerrojo* a 0; en este momento solo uno de los procesos que esperan obtendrá acceso a su sección critica. La elección del proceso depende del que ejecute la instrucción Comparar y Fijar a continuación.

Instrucción Intercambiar La instrucción Intercambiar se puede definir como sigue:

```
procedure intercambiar (var r: registro; var m: memoria);
var temp;
begin
        temp := m;
        m := r;
        r := temp
end.
```

Esta instrucción intercambia el contenido de un registro con el de una posición de memoria. Durante la ejecución de la instrucción, se bloquea el acceso a la posición de memoria de cualquier otra instrucción que haga referencia a la misma posición

La figura 4.8b muestra un protocolo de exclusión mutua que se basa en el empleo de esta instrucción. Se da un valor 0 inicial a una variable compartida *cerrojo.* El único proceso que puede entrar en su sección critica es el que encuentre *cerrojo* igual a 0. Poniendo *cerrojo* a 1 se excluye a todos los demás procesos de entrar en la sección crítica. Cuando un proceso abandona su sección crítica, vuelve a poner *cerrojo* a 0, permitiendo que otros procesos obtengan acceso a su sección crítica.

```
program exclusión_mutua;
const n=...; (*número de procesos*);
var cerrojo: entero;
procedure P (i: entero);
begin
   repeat
      repeat { nada }until TS (cerrojo);
         < sección crítica >;
         cerrojo := 0;
         < resto >
      forever
end;
begin (* programa principal *)
   cerrojo := 0;
   parbegin
      P(1);
      P(2);
      ...
      P(n);
   parend
end.
```

(a) Instrucción Comparar y Fijar

```
program exclusión_mutua;
const n=...; (*número de procesos*);
var cerrojo: entero;
procedure P (i: entero);
var clave_i: entero;
begin
   repeat
      clave_i := 1;
      repeat intercambiar (clave_i, cerrojo) until clave_i = 0;
      < sección crítica >;
      intercambiar (clave_i, cerrojo);
      < resto >
   forever
end;
begin (* programa principal *)
   cerrojo := 0;
   parbegin
      P(1);
      P(2);
      ...
      P(n);
   parend
end..
```

(b) Instrucción Intercambiar

FIGURA 4.8 Soporte de hardware para la exclusión mutua

Propiedades de las soluciones con instrucciones de máquina

El uso de instrucciones especiales de la máquina para hacer cumplir la exclusión mutua tiene varias ventajas:

- Es aplicable a cualquier número de procesos en sistemas con memoria compartida, tanto de monoprocesador como de multiprocesador.
- Es simple y fácil de verificar.
- Puede usarse para disponer de varias secciones críticas; cada sección crítica puede definirse con su propia variable.

Algunas desventajas importantes son las siguientes:

- Se emplea espera activa. Así pues, mientras un proceso está esperando para acceder a la sección critica, continua consumiendo tiempo del procesador.
- Puede producirse inanición. Cuando un proceso abandona la sección crítica y hay más de un proceso esperando, la selección es arbitraria. Así pues, se podría denegar el acceso a algún proceso indefinidamente.
- Puede producirse interbloqueo. Supóngase la siguiente escena en un sistema monoprocesador. El proceso P1 ejecuta una instrucción especial (sea TS o Intercambiar) y entra en su sección critica. P1 es interrumpido para dar el procesador a P2, que tiene mayor prioridad. Si P2 intenta ahora usar el mismo recurso que P1, se le negará el acceso por el mecanismo de exclusión mutua. De este modo, P2 entrará en un bucle de espera activa. Sin embargo, P1 nunca será expedido porque su prioridad es menor que la de otro proceso listo, esto es, P2.

Debido a estos inconvenientes tanto de las soluciones por software como por hardware, es necesario buscar otros mecanismos.

15.1.5 mecanismos que se usan en el sistema operativo y en los lenguajes de programación

15.1.5.1 Semaforos:

Para lograr el efecto deseado, se pueden contemplar los semáforos como variables que tienen un valor entero sobre el que se definen las tres operaciones siguientes:

1. Un semáforo puede inicializarse con un valor no negativo.
2. La operación *wait* decrementa el valor del semáforo. Si el valor se hace negativo, el proceso que ejecuta el *wait* se bloquea.
3. La operación *signal* incrementa el valor del semáforo. Si el valor no es positivo, se desbloquea a un proceso bloqueado por una operación *wait.*

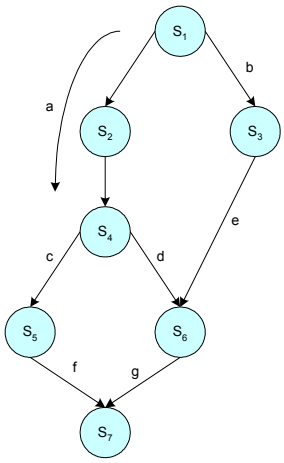
Aparte de estas tres operaciones, no hay forma de examinar o manipular los semáforos.

La figura 4.9 propone una definición más formal de las primitivas de los semáforos. Las pri-mitivas *wait* y *signal* se suponen atómicas, es decir, no pueden ser interrumpidas y cada rutina puede considerarse como un paso indivisible. En la figura 4.10 se define una versión más limi-tada, conocida como *semáforo binario.* Un semáforo binario solo puede tomar los valores 0 y 1. En principio, los semáforos binarios son más sencillos de implementar y puede demostrarse que tienen la misma potencia de expresión que los semáforos generales (ver problema 4.14).

Tanto en los semáforos como en los semáforos binarios se emplea una cola para mantener los procesos esperando en el semáforo. La definición no dicta el orden en el que se quitan los proce-sos de dicha cola. La política más equitativa es la FIFO: el proceso que ha estado bloqueado du-rante más tiempo se libera de la cola. La única exigencia estricta es que un proceso no debe que-dar retenido en la cola de un semáforo indefinidamente porque otros procesos tengan preferencia.

15.1.5.1.1 Ejemplo:

```
VAR a, b, c, d, e, f, g SEMAFORO;
```




```
INIT (a, 0);
INIT (b, 0);
INIT (c, 0);
INIT (d, 0);
INIT (e, 0);
INIT (f, 0);
INIT (g, 0);
COBEGIN

BEGIN S1; V(a); V(b); END
BEGIN P(a); S2; S4; V(c); V(d); END
BEGIN P(b); S3; V(e); END
BEGIN P(c); S5; V(f); END
BEGIN P(d); P(e); S6; V(g); END
BEGIN P(f); P(g); S7; END

COEND
```

Otra forma, es tratando de ahorrar semáforos.

```
VAR d SEMAFORO;
INIT (d, 0);

S1;

COBEGIN

BEGIN S2; S4; V(d); S5; END
BEGIN S3; V(d); END
BEGIN P(d); P(d); S6; END

COEND

S7;
```

Se debe crear mutua exclusión entre P y V, para ello, se usa busy waiting, ya que los códigos de P y V son cortos, y la espera es muy pequeña.

La lista de procesos en espera (cuando hago P), se puede implementar de diferentes formas.

15.1.5.2 6.4.3 Bloqueos mutuos e inanición

Se puede llegar a dar el caso en que dos o más procesos esperen indefinidamente la ocurrencia de un suceso. Estos procesos están en bloqueo mutuo.

15.1.5.2.1 Ejemplo:

P_1 ejecuta Espera(S), y P_2 ejecuta Espera(Q), P_2 no va a poder continuar hasta que P_1 no ejecute Señal(Q), y de la misma forma, P_1 no puede continuar hasta que P_2 ejecute Señal (S). Los dos procesos están en bloqueo mutuo.

P_1	P_2
Espera(S)	Espera(Q)
Espera(Q)	Espera(S)
.	.
.	.
.	.
Señal(Q)	Señal(S)
Señal(S)	Señal(Q)

Otro problema es la *inanición* que es cuando algunos procesos esperan indefinidamente dentro del semáforo. Puede suceder si la lista de espera del semáforo es LIFO.

```
wait(s):
        repeat { nothing } until testset(s.flag);
        s.count := s.count - 1;
        if s.count < 0
            then begin
                place this process in s.queue;
                block this process (must also set s.flag
to 0)
                end
            else s.flag := 0;

signal(s):
        repeat { nothing } until testset(s.flag);
        s.count := s.count + 1;
        if s.count ≤ 0
            then begin
                remove a process P from s.queue;
                place process P on ready list
                end;
        s.flag := 0;
```

(a) Testset Instruction

```
wait(s):
        inhibit interrupts;
        s.count := s.count - 1;
        if s.count < 0
            then begin
                place this process in s.queue;
                block this process and allow interrupts
                end
            else allow interrupts;

signal(s):
        inhibit interrupts;
        s.count := s.count + 1;
        if s.count ≤ 0
            then begin
                remove a process P from s.queue;
                place process P on ready list
                end;
        allow interrupts;
```

(b) Interrupts

Figure 5.17 Two Possible Implementations of Semaphores

```
type  binary semaphore =     record
                               value: (0,1);
                               queue: list of process
                             end;
var s: binary semaphore;

waitB(s):
        if s.value =  1
           then
              s.value = 0
           else begin
              place this process in s.queue;
              block this process
           end;

signalB(s):
        if s.queue is empty
           then
              s.value := 1
           else begin
              remove a process P from s.queue;
              place process P on ready list
           end;
```

Figure 5.7 A Definition of Binary Semaphore Primitives

```
type semáforo = record
                    contador: entero;
                    cola: list of proceso
                end;
var s: semáforo;
wait(s):
    s.contador := s.contador – 1;
    if s.contador < 0
        then begin
            poner este proceso en s.cola;
            bloquear este proceso
        end;
signal(s):
    s.contador := s.contador + 1;
    if s.contador ≤ 0
        then begin
            quitar un proceso P de s.cola;
            poner el proceso P en la cola de listos
        end;
```

FIGURA 4.9 Una definición de las primitivas de los semáforos

```
program      mutualexclusion;
const n = . . . ; (*number of processes*);
var      s: semaphore (:=1);
procedure P(i: integer);
begin
   repeat
   wait(s);
   < critical section >;
   signal(s);
   < remainder >
   forever
end;
begin (* main program *)
   parbegin
      P(1);
      P(2);
      . . .
      P(n)
   parend
end.
```

Figure 5.9 Mutual Exclusion Using Semaphores

15.1.5.2.2 Ej.- Productor – Consumidor

15.1.5.2.2.1 buffer ilimitado:

En términos abstractos, se puede definir la función del productor como sigue:

productor:
repeat
producir elemento v;

```
                b[ent] := v;
                ent := ent +1
        forever;
end productor;
```

Del mismo modo, la función del consumidor toma la siguiente forma:

```
consumidor:
        repeat
                while ent ≤ sal do { nada };
                        w := b[sal];
                        sal := sal +1;
                        consumir elemento w
        forever;

end productor;
```

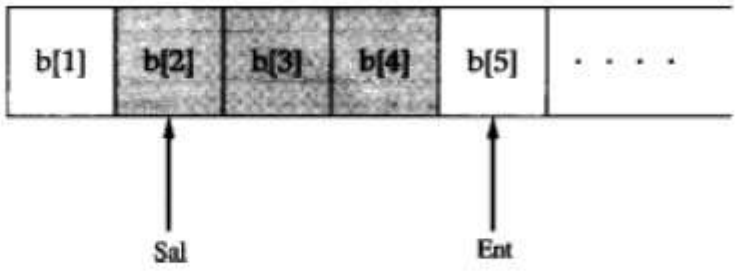


FIGURA 4.13 Buffer ilimitado en el problema del productor/consumidor

N=ENT-SAL ES LA CANTIDAD DE ELEMENTOS EN EL BUFFER
SEMAFORO S, SE UTILIZA PARA HACER CUMPLIR LA EXCLUSION MUTUA.
SEMAFORO RETRASO SE USA PARA OBLIGAR AL CONSUMIDOR A ESPERAR SI EL BUFFER ESTA VACÍO.

```
program    producerconsumer;
var        n: integer;
           s: (*binary*) semaphore (:= 1);
           delay: (*binary*) semaphore (:= 0);
procedure producer;
begin
   repeat
      produce;
      waitB(s);
      append;
      n := n + 1;
      if n=1 then signalB(delay);
      signalB(s)
   forever
end;
procedure consumer;
begin
   waitB(delay);
   repeat
      waitB(s);
      take;
      n := n - 1;
      signalB(s);
      consume;
      if n=0 then waitB(delay)
   forever
end;
begin (*main program*)
   n := 0;
   parbegin
      producer; consumer
   parend
end.
```

Figure 5.13 An Incorrect Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

hay un defecto en este programa. Cuando el consumidor haya agotado el buffer, necesita restaurar el semáforo *retraso,* así que se ye obligado a esperar hasta que el productor haya colocado mas elementos en el buffer. Este es el propósito de la sentencia: if *n*= 0 **then** *waitB(retraso)*. Considérese la situación representada en la tabla 4.2. En la

línea 6, el consumidor falla al ejecutar la operación *waitB*. El consumidor en realidad agota el buffer y pone *n* a 0 (línea 4), pero el productor ha incrementado *n* antes de que el consumidor pueda comprobarlo en la línea 6. El resultado es un *signalB* que no se corresponde con ningún *waitB* anterior. El valor de -1 de *n* en la línea 9 significa que el consumidor ha consumido un elemento del buffer que no existe. No bastarla simplemente con **mover la sentencia condicional dentro de la sección critica** de consumidor porque **esto podría llevar a interbloqueo** (por ejemplo, en la línea 3).

Un arreglo al problema es introducir una variable auxiliar que pueda recibir un valor dentro de la sección crítica del consumidor para un uso posterior, como se muestra en la figura 4.15. Un recorrido cuidadoso de la lógica del programa convencerá de que no se puede producir interbloqueo.

TABLA 4.2 Posible situación para el programa de la figura 4.15

	Acción	*n*	*Retraso*
1	Inicialmente	0	0
2	Productor: sección critica	1	1
3	Consumidor: waitB(retraso)	1	0
4	Consumidor: sección critica	0	0
5	Productor: sección critica	1	1
6	Consumidor: if *n* = 0 then waitB(retraso)	1	1
7	Consumidor: sección critica	0	1
8	Consumidor: if *n* = 0 then waitB(retraso)	0	0
9	Consumidor: sección critica	-1	0

```
program    producerconsumer;
var        n: integer;
           s: (*binary*) semaphore (:= 1);
           delay: (*binary*) semaphore (:= 0);
procedure producer;
begin
   repeat
      produce;
      waitB(s);
      append;
      n := n + 1;
      if n=1 then signalB(delay);
      signalB(s)
   forever
end;
procedure consumer;
var   m: integer; (* a local variable *);
begin
   waitB(delay);
   repeat
      waitB(s);
      take;
      n := n - 1;
      m := n;
      signalB(s);
      consume;
      if m=0 then waitB(delay)
   forever
end;
begin (*main program*)
   n := 0;
   parbegin
      producer; consumer
   parend
end.
```

Figure 5.13 A Correct Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

```
program    producerconsumer;
var        n: semaphore (:= 0);
           s: semaphore (:= 1);
procedure producer;
begin
   repeat
      produce;
      wait(s);
      append;
      signal(s);
      signal(n)
   forever
end;
procedure consumer;
begin
   repeat
      wait(n);
      wait(s);
      take;
      signal(s);
      consume
   forever
end;
begin (*main program*)
   parbegin
      producer; consumer
   parend
end.
```

Figure 5.14 A Solution to the Infinite-Buffer Producer/Consumer Problem Using Semaphores

15.1.5.2.2.2 buffer limitado

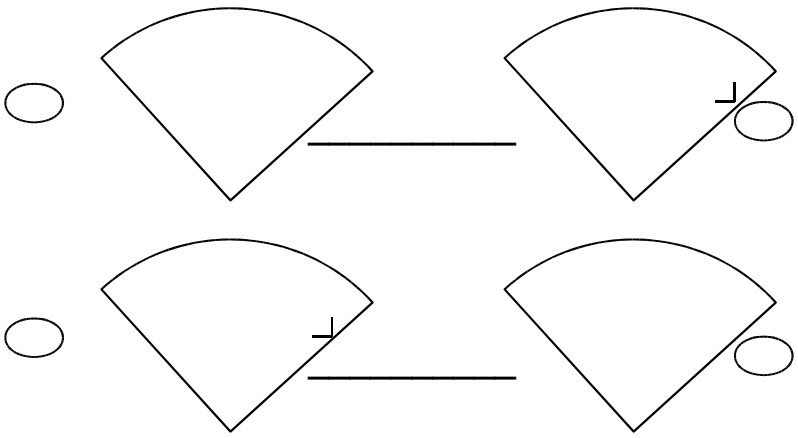
Tenemos dos procesos – uno que produce datos
- uno que consume los datos a la vez
(ambos procesos van a distintas velocidades).
Entre ambos procesos existirá un buffer.
→ hay reglas:
→ buffer finito
→ el consumidor cuando el buffer está vacía debe quedar bloqueado hasta que hayan datos
→ el productor si el buffer está lleno debe aguardar bloqueado a que se libere espacio en el buffer

problema del productor - consumidor

1 o varios procesos 1 o varios procesos

todos los procesos trabajan a distintas velocidades

prod cons



productor:

```
repeat
        producir elemento v;
        while ((ent + 1) mod n = sal) do { nada };
                b[ent] := v;
                ent := (ent + 1) mod n
forever;
```

consumidor:

```
repeat
```

```
            while ent = sal do { nada };
                    w := b[sal];
                    sal := (sal + 1) mod n;
                    consumir elemento w
        forever;
```

→ solución del problema utilizando semáforos.-

```
program  prod – cons
var l , n : semáforo;
        buff : buffer;
procedure produce
        loop
                producir (a);
                p (l);
                p (s);
                agrega_buff (a) ;
                v (s);
                v (n);
      endloop
```

cuando el semáforo n = 10 significa que hay 10 elementos en el buffer, l significa cantidad de espacios libres en el buffer.

```
Procedure consume
Loop
        p (n);
        p (s);
        Saca_buff
        v (s);
        v (l);
                consumir
endloop


begin
        init ( n, 0)
        init ( l , tam_buff);
        init ( s, 1)
        cobegin
                produce;
                consume;
        coend
end
```

la producción y consumición aquí si se puede estar haciendo simultáneamente (siempre que halla variables globales, hay problemas de mutua exclusión).

```
program    boundedbuffer;
const sizeofbuffer = . . .;
var        s:  semaphore (:= 1);
           n: semaphore (:= 0);
           e: semaphore (:= sizeofbuffer);
procedure producer;
begin
   repeat
      produce;
      wait(e);
      wait(s);
      append;
      signal(s);
      signal(n)
   forever
end;
procedure consumer;
begin
   repeat
      wait(n);
      wait(s);
      take;
      signal(s);
      signal(e);
      consume
   forever
end;
begin (*main program*)
   parbegin
      producer; consumer
   parend
end.
```

Figure 5.16 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Semaphores

15.1.5.2.3 Problema de los lectores escritores:

Prioridad a los lectores

La figura 4.29a es una solución que utiliza semáforos, mostrando un caso de un lector y otro de un escritor; la solución no cambia para el caso de varios lectores y escritores. El proceso escritor es sencillo. El semáforo *esem* se usa para respetar la exclusión mutua. Mientras que un escritor está accediendo a los datos compartidos, ningún otro escritor y ningún lector podrá acceder. El proceso lector también usa el semáforo *esem* para respetar la exclusión mutua. Sin embargo, para que se permitan varios lectores, hace falta que, cuando no haya ninguno, el primer lector que lo intente tenga que esperar en *esem.* Cuando ya hay al menos un lector, los lectores posteriores no necesitan esperar antes de entrar. La variable global *contlect* se utiliza para mantener el número de lectores y el semáforo *x* se utiliza para asegurar que *contlect* se actualiza correctamente.

```
program readersandwriters;
var   readcount: integer;
      x, wsem: semaphore (:= 1);
procedure reader;
begin
   repeat
      wait (x);
         readcount := readcount + 1;
         if readcount = 1 then wait (wsem);
      signal (x);
      READUNIT;
      wait (x);
         readcount := readcount - 1;
         if readcount = 0 then signal (wsem);
      signal (x)
   forever
end;
procedure writer;
begin
   repeat
      wait (wsem);
         WRITEUNIT;
      signal (wsem)
   forever
end;
begin
   readcount := 0;
   parbegin
      reader;
      writer
   parend
end.
```

Figure 5.28 A Solution to the Readers/Writers Problem Using Semaphores: Readers have priority

Prioridad a los escritores En la solución anterior, los lectores tienen prioridad. Una vez que un lector ha comenzado a acceder a los datos, es posible que los lectores mantengan el control del área de datos mientras que haya al menos uno leyendo. Por tanto, los escritores están sujetos a

inanición. La figura 4.29b muestra una solución que garantiza no permitir acceder a los datos a ningún nuevo lector una vez que, al menos, un escritor haya declarado su deseo de escribir. Para los escritores, se añaden los siguientes semáforos y variables al ya definido: Un semáforo *Isem* que inhibe todas las lecturas mientras haya al menos un escritor que desee acceder a los datos. • Una variable *contesc* que controla la activación de *Isem.* • Un semáforo y que controla la actualización de *contesc.* Para los lectores, se necesita un semáforo adicional. No debe permitirse que se construya una cola grande sobre *Isem,* pues los escritores no serían capaces de saltarla. Por tanto, sólo se per-mite a un lector ponerse en cola en *Isem* y todos los demás lectores deben ponerse en cola en un semáforo z inmediatamente antes de esperar en *Isem.* La tabla 4.5 resume las posibilidades.

TABLA 4.5 Estado de las colas de procesos para el programa de Ia figura 4.29b

Sólo lectores en el sistema	• Activar *esem* • Sin colas
Sólo escritores en el sistema	• Activar *esem* y *Isem* • Los escritores se encolan en *esem*
Lectores y escritores con un lector primero	• *Esem* activado por un lector • *Isem* activado por un escritor • Todos los escritores se ponen en cola en *esem* • Un lector se pone en cola en *Isem* • Los otros lectores se ponen en cola en *z*
Lectores y escritores con un escritor primero	• Los escritores activan *esem* • Los lectores activan *Isem* • Todos los escritores se ponen en cola en esem • Un lector se pone en cola en *Isem* • Los otros lectores se ponen en cola en *z*

```
program readersandwriters;
var readcount, writecount: integer;
        x, y, z, wsem, rsem: semaphore (:= 1);
procedure reader;
begin
    repeat
        wait (z);
            wait (rsem);
                wait (x);
                    readcount := readcount + 1;
                    if readcount = 1 then wait (wsem);
                signal (x);
            signal (rsem);
        signal (z);
        READUNIT;
        wait (x);
            readcount := readcount - 1;
            if readcount = 0 then signal (wsem);
        signal (x)
    forever
end;
procedure writer;
begin
    repeat
        wait (y);
            writecount := writecount + 1;
            if writecount = 1 then wait (rsem);
        signal (y);
        wait (wsem);
        WRITEUNIT;
        signal (wsem);
        wait (y);
            writecount := writecount - 1;
            if writecount = 0 then signal (rsem);
        signal (y)
    forever
end;
begin
    readcount, writecount := 0;
    parbegin
        reader;
        writer
    parend
end.
```

Figure 5. 29 A Solution to the Readers/Writers Problem Using Semaphores: **Writers have priority**

15.1.5.2.4 Problema de la barberia:

barbero()
...
..
..
signal(terminado)
wait(dejar sila) <---------- importante porque el cliente puede no liberar el recurso.
signal(sillabatbero) Tengo que asegurarme que lo libera, antes de darle permiso para entrar a otro

wait (terminado)
levantarse de la silla del barbero
signal(dejar_silla)

al incializar el semaforo coord con 3 puedo tirar los procesos barbero (3) y cajero (1) y asegurarme que se cobra cuando no se corta (uno libre por lo menos).

```
program  barbershop1;
var      max_capacity: semaphore (:= 20);
         sofa: semaphore (:= 4);
         barber_chair, coord: semaphore (:= 3);
         cust_ready, finished, leave_b_chair, payment, receipt: semaphore (:= 0);

procedure customer;
begin
    wait(max_capacity);
    enter shop;
    wait(sofa);
    sit on sofa;
    wait(barber_chair);
    get up from sofa;
    signal(sofa);
    sit in barber chair;
    signal(cust_ready);
    wait(finished);
    leave barber chair;
    signal(leave_b_chair);
    pay;
    signal(payment);
    wait(receipt);
    exit shop;
    signal(max_capacity)
end;

procedure barber;
begin
    repeat
        wait(cust_ready);
        wait(coord);
        cut hair;
        signal(coord);
        signal(finished);
        wait(leave_b_chair);
        signal(barber_chair);
    forever
end;

procedure cashier;
begin
    repeat
        wait(payment);
        wait(coord);
        accept pay;
        signal(coord);
        signal(receipt);
    forever
end;

begin (*main program*)
    parbegin
        customer; . . . 50 times; . . . customer;
        barber; barber; barber;
        cashier
    parend
end.
```

Figure 5.19 An Unfair Barbershop

```
program  barbershop2;
var      max_capacity: semaphore (:= 20);
         sofa: semaphore (:= 4);
         barber_chair, coord: semaphore (:= 3);
         mutex1, mutex2: semaphore (:=1);
         cust_ready, leave_b_chair, payment, receipt: semaphore (:= 0);
         finished: array[1..50] of semaphore (:=0);
         count: integer;

procedure customer;
var custnr: integer;
begin
    wait(max_capacity);
    enter shop;
    wait(mutex1);
    count := count + 1;
    custnr := count;
    signal(mutex1);
    wait(sofa);
    sit on sofa;
    wait(barber_chair);
    get up from sofa;
    signal(sofa);
    sit in barber chair;
    wait(mutex2);
    enqueue1(custnr);
    signal(cust_ready);
    signal(mutex2);
    wait(finished[custnr]);
    leave barber chair;
    signal(leave_b_chair);
    pay;
    signal(payment);
    wait(receipt);
    exit shop;
    signal(max_capacity)
end;

procedure barber;
var b_cust: integer;
begin
    repeat
        wait(cust_ready);
        wait(mutex2);
        dequeue1(b_cust);
        signal(mutex2);
        wait(coord);
        cut hair;
        signal(coord);
        signal(finished[b_cust]);
        wait(leave_b_chair);
        signal(barber_chair);
    forever
end;

procedure cashier;
begin
    repeat
        wait(payment);
        wait(coord);
        accept pay;
        signal(coord);
        signal(receipt);
    forever
end;

begin (*main program*)
    count := 0;
    parbegin
        customer; . . . 50 times; . . . customer;
        barber; barber; barber;
        cashier
    parend
end.
```

Figure 5.20 A Fair Barbershop

15.1.5.2.5 El problema de la cena de los filosofos:

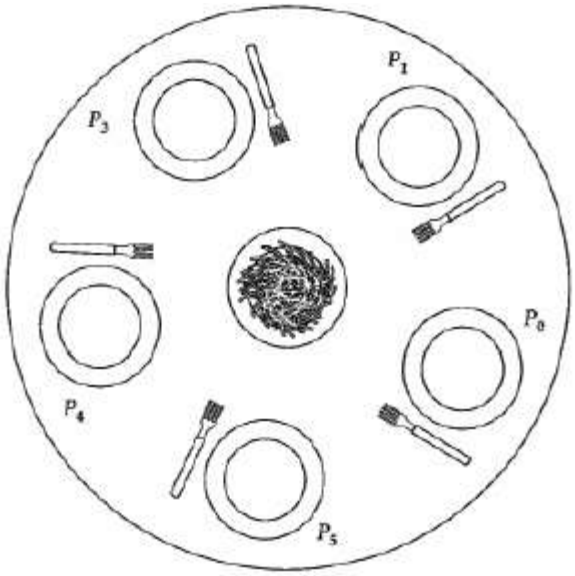
Érase una vez cinco filósofos que vivían juntos. La vida de cada filósofo consistía principal-mente en pensar y comer y, tras años de pensar, todos los filósofos se habían puesto de acuerdo en que la única comida que contribuía a sus esfuerzos pensadores eran los espaguetis.

Los preparativos de la comida eran simples (figura 5.6): una mesa redonda en la que había una gran fuente de espaguetis, cinco platos, uno para cada filósofo y cinco tenedores. Un filósofo que quiera comer irá a su lugar asignado en la mesa y, usando los dos tenedores de cada lado del plato, cogerá los espaguetis y se los comerá. El problema es el siguiente: Inventar un ritual (algoritmo) que permita comer a los filósofos. El algoritmo debe satisfacer la exclusión mutua (dos filósofos no pueden emplear el mismo tenedor a la vez), además de evitar el interbloqueo y la inanición (en este caso, este último término tiene significado literal además de algorítmico).

Este problema, propuesto por Dijkstra, puede no parecer importante o relevante por sí mismo. Sin embargo, sirve para ilustrar los problemas básicos del interbloqueo y la inanición. Es más, intentar desarrollar una solución revela muchas de las dificultades de la programación concurrente (véase, por ejemplo, [GING90]). Por consiguiente, este problema es un caso de prueba estándar para examinar soluciones a la sincronización.

La figura 5.7 sugiere una solución por medio de semáforos. Cada filósofo toma primero el tenedor de su izquierda y, después, el de su derecha. Cuando un filósofo termina de comer, devuelve los dos tenedores a la mesa. Esta solución, desafortunadamente, produce interbloqueo: Si todos los filósofos están hambrientos al mismo tiempo, todos se sientan, todos cogen el tenedor de su izquierda y todos intentan tomar el otro tenedor, que no estará. En esta situación informal, todos los filósofos pasan hambre.

Para superar el riesgo de interbloqueo, se podrían adquirir cinco tenedores adicionales (una solución más saludable) o enseñar a los filósofos a comer espaguetis con sólo un tenedor. Como otra solución posible, se podría considerar incorporar un **sirviente que permita pasar sólo a cuatro filósofos a la vez al comedor.** Con un máximo de cuatro filósofos sentados, al menos uno de los filósofos tendrá acceso a los dos tenedores. La figura 5.8 muestra esta solución, de nuevo con semáforos. Esta solución está libre de interbloqueo e inanición.



```
program cena_filósofos;
var  tenedor: array [0 ... 4] of semáforo (:= 1);
     i: entero;
procedure filósofo (i: entero);
begin
 repeat
     pensar;
     wait (tenedor[i]);
     wait (tenedor[(i+1) mod 5]);
     comer;
     signal (tenedor[(i+1) mod 5]);
     signal (tenedor[i])
 forever
end;
begin
 parbegin
     filósofo (0);
     filósofo (1);
     filósofo (2);
     filósofo (3);
     filósofo (4);
 parend
end.
```

FIGURA 5.7 Una primera solución al problema de la cena de los filósofos

```
program cena_filósofos;
var tenedor: array [O ... 4] of semáforo (:= 1);
habitación: semáforo (:= 4);
i: entero;
procedure filósofo (i: entero);
begin
repeat
        pensar;
        wait (habitación);
        wait (tenedor[i]);
        wait (tenedor[(i+l) mod 5]);
        comer;
        signal (tenedor[(i+l) mod 5]);
        signal (tenedor[i]);
        signal (habitación)
forever
end;

begin
```

parbegin
filósofo (0);
filósofo (1);
filósofo (2);
filósofo (3);
filósofo (4);
parend end.

FIGURA 5.8 Segunda solución al problema de la cena de los filósofos

Este ejercicio demuestra la sutileza del problema de la cena de los filósofos y la dificultad de escribir programas correctos usando semáforos. La solución siguiente al problema de la cena de los filósofos, escrita en C, se encuentra en el texto de Tanenbaum [TANE87]:

```
#define N                5          /* número de filósofos */
#define IZQUIERDA        (i -1)     /* número vecino izquierdo de i */
MOD N
#define DERECHA          (i + 1)    /* número vecino derecho de i */
MOD N
#define MEDITANDO        0          /* el filósofo está meditando */
#define HAMBRIENTO       1          /* el filósofo intenta tomar los tenedores */
#define COMIENDO         2          /* el filósofo está comiendo */
typedef int semáforo;               /* los semáforos son una clase de enteros */
int estado [N];                     /* vector con el estado de cada uno */
semáforo mutex = 1;                 /* exclusión mutua en regiones críticas */
```

```
semáforo s[N];                          /* un semáforo por filósofo */
filósofo (i)
int i;                                  /* número del filósofo, 0 a N–1  */
{
   while (TRUE) {                       /* bucle infinito */
         meditar ( );                   /* el filósofo está meditando */
         tomar_tenedores (i);           /*toma dos tenedores o se bloquea */
         comer ( );                     /* ñam-ñam, espaguetis */
         dejar_tenedores (i);           /* deja ambos tenedores en la mesa */
   }
}

tomar_tenedores (i)
int i;                                  /* número del filósofo, 0 a N–1 */
{
   wait (mutex);                        /* entra en región crítica */
   estado[i] = HAMBRIENTO;              /* recuerda que filósofo i está hambriento */
   comprobar (i);                       /* intenta tomar 2 tenedores */
   signal (mutex);                      /* sale de región crítica */
   wait (s[i]);                         /* se bloquea si no se tienen los tenedores */
}

dejar_tenedores (i)
int i;                                  /* número del filósofo, 0 a N–1 */
{
   wait (mutex);                        /* entra en región crítica */
   estado[i] = MEDITANDO;               /* el filósofo i ha terminado de comer */
   comprobar (IZQUIERDA);               /* ¿puede el de la izquierda comer ahora? */
   comprobar (DERECHA);                 /* ¿puede el de la derecha comer ahora? */
   signal (mutex);                      /* sale de región crítica */
}

comprobar (i)
int i;                                  /* número del filósofo, 0 a N–1 */
{
   if (estado[i] == HAMBRIENTO && estado[IZQUIERDA] != COMIENDO &&
       estado[DERECHA] != COMIENDO) {estado[i] = COMIENDO;
       signal (s[i]);
   }
}
```

a) Describir en pocas palabras el funciona-miento de esta solución.
b) Tanenbaum afirma que: "La solución [...] es correcta y permite el máximo paralelismo para un número arbitrario de filósofos". Aunque esta solución previene el interbloqueo, no es correcta porque la inanición es posible. Demuéstrese con un contraejemplo. Indicación: Considérese el caso de cinco filósofos. Supóngase que son filósofos glotones: No pasan

apenas tiempo pensando. Tan pronto como un filósofo ha terminado un tumo de comida, le entra el hambre casi de inmediato. Considérese entonces una configuración en la que dos filósofos están comiendo y los otros tres bloqueados y hambrientos.

solucion:

6.15 a. When a philosopher finishes eating, he allows his left neighbor to proceed if possible, then permits his right neighbor to proceed. The solution uses an array, *state*, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks). A philosopher may move only into the eating state if neither neighbor is eating. Philosopher i's neighbors are defined by the macros LEFT and RIGHT.

b. This counterexample is due to [GING90]. Assume that philosophers P0, P1, and P3 are waiting with hunger while philosophers P2 and P4 dine at leisure. Now consider the following admittedly unlikely sequence of philosophers' completions of their suppers.

EATING	HUNGRY
4 2	0 1 3
2 0	1 3 4
3 0	1 2 4
0 2	1 3 4
4 2	0 1 3

Each line of this table is intended to indicate the philosophers that are presently eating and those that are in a state of hunger. The dining philosopher listed first on each line is the one who finishes his meal next. For example, from the initial configuration, philosopher P4 finishes eating first, which permits P0 to commence eating. Notice that the pattern folds in on itself and can repeat forever with the consequent starvation of philosopher P1.

15.1.5.3 Monitores:

Nota.- cuando utilizamos monitores no debemos utilizar variables globales.

15.1.5.3.1 Implementar semáforos no binarios usando monitores:

```
type semáforo = MONITOR
   var  cont : integer
                ocupado : condition
        procedure P;
                if ( cont = 0 )
                        ocupado . WAIT;
                cont := cont – 1;
        endP;
        procedure V;
                if ( NOEMPTY ( ocupado ) ) then
                        ocupado . SIGNAL;
                else cont := cont + 1;
        endV
        begin   Monitor
                cont := n
        end     monitor
```

15.1.5.3.2 implementación de semáforos binarios utilizando monitores

```
type  semáforo = MONITOR
         var   busy : boolean;
                nobusy : condition;
         Procedure  P
                if busy then nobusy.WAIT
                busy := true
        EndP

        Procedure V
                busy := false
                nobusy.SIGNAL
        EndV
        Begin
                busy := false;
        End MONITOR
```

} aquí false=1, true=0

← inicialización del semáforo

Todo lo que se podía resolver con semáforos se podrá resolver con monitores ➔ semáforos equivalente monitores

15.1.5.3.3 Productor consumidor:

```
program producerconsumer
    monitor boundedbuffer;
            buffer: array [0 . . N] of char;          {space for N items}
            nextin, nextout: integer;                 {buffer pointers}
            count: integer;                           {number of items in buffer}
            notfull, notempty: condition;             {for synchronization}

    procedure append (x: char);
            begin
                if count = N then cwait(notfull);     {buffer is full; avoid overflow}
                buffer[nextin] := x;
                nextin := nextin + 1 mod N;
                count := count + 1;                   {one more item in buffer}
                csignal(notempty);                    {resume any waiting consumer}
            end;
    procedure take (x: char);
            begin
                if count = 0 then cwait(notempty);    {buffer is empty; avoid underflow}
                x := buffer[nextout];
                nextout := nextout + 1 mod N;
                count := count - 1;                   {one fewer item in buffer}
                csignal(notfull);                     {resume any waiting producer}
            end;
    begin                                             {monitor body}
            nextin := 0; nextout := 0; count := 0     {buffer initially empty}
    end;

procedure producer;
var         x: char;
begin
    repeat
        produce(x);
        append(x);
    forever
end;
procedure consumer;
var         x: char;
begin
    repeat
        take(x);
        consume(x);
    forever
end;
begin (*main program*)
    parbegin
        producer; consumer
    parend
end.
```

Figure 5.22 A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor

15.1.5.3.4 LECTORES – ESCRITORES

ahora vamos a incorporar el que podamos averiguar si existe algún proceso bloqueado por esa condición.

```
Procedure R_W
Monitor READ-WRITE
var
        readers : integer
        writing : boolean
        oktoread : condition ; oktowrite : condition;

  Procedure START-READ
                                                    me indican cuantos hay
                                                    esperando para escribir (si
                                                    es vacía o no)
                if writing or nonempty (oktowrite)
                        wait oktoread
                endif
                readers ++;
                signal.oktoread;                    si se coloca esto se le
  end                                               estaría quitando la prioridad
  Procedure END_READERS          a los lectores
                readers - -
                if (readers = 0 )
                  signal.oktowrite
                endif
  end
  Procedure START-WRITE
                if readers <> 0 or writing
                        wait.oktowrite
                endif
                writing := TRUE
  end
  Procedure END-WRITE
                writing := FALSE;
                if NONEMPTY (oktoread)
                        signal.oktoread
                else
                        signal.oktowrite
  end

 Begin   monitor
        readers = 0
        writing = FALSE
 End

 Procedure LECTOR
                while TRUE do
```

```
                    START_READ;
                    READ;
                    END_READ;
                End
 End
```

Nota.- siempre que haya prioridades esta presente el problema de posposición indefinida.

15.1.5.3.5 Problema de los filosofos:

```
procedure FILOSOFOS;
 MONITOR comedor
 var  comensales : integer;
        lleno : condition
procedure ENTRADA;
begin
        if ( comensales = 4 ) then
                lleno . WAIT;
        comensales := comensales + 1 ;
end;
procedure SALIDA;
begin
        comensales - -;
        if NONEMPTY ( lleno ) then
                lleno . SIGNAL;
end;

begin   MONITOR
        comensales = 0;
end;   MONITOR
MONITOR TENEDOR;
var  enuso : boolean;
     ocupado : condition;
procedure LEVANTA;
begin
        if (enuso) then
                ocupado . WAIT ;
        enduso := TRUE;
end;
procedure DEJA;
begin
        enuso := FALSE;
        ocupado . SIGNAL;
end;
begin  MONITOR
        enuso := FALSE;
end  MONITOR


procedure FILOSOFO ( i : in integer);
```

```
  var der , izq : integer;
 begin
        izq := i ;
        der := ( i + 1 ) mod 5;
        loop
                COMEDOR . ENTRADA;
                TENEDOR izq . LEVANTA;
                TENEDOR der . LEVANTA;
                COME;
                TENEDOR der . DEJA;
                TENEDOR izq . DEJA;
                COMEDOR . SALIDA;
        endloop;
end   FILOSOFO
begin  FILOSOFOS
        cobegin
                FILOSOFO ( 0 );
                FILOSOFO ( 1 );
                …….
                FILOSOFO ( 4 );
        coend;
end;  FILOSOFOS
```

15.1.5.4 Mensajería:

La funcionalidad real del paso de mensajes se ofrece, normalmente, por medio de un par de primitivas:

- *send (destino, mensaje)*
- *receive (origen, mensaje)*

Este es el conjunto mínimo de operaciones necesario para que los procesos puedan dedicarse al paso de mensajes.

Son habituales las siguientes tres combinaciones:

• ***Envío bloqueante, recepción bloqueante:*** Tanto el emisor como el receptor se bloquean hasta que se entrega el mensaje; esta técnica se conoce como *rendezvous.* Esta combinación permite una fuerte sincronización entre procesos.

• ***Envío no bloqueante, recepción bloqueante:*** Aunque el emisor puede continuar, el receptor se bloquea hasta que llega el mensaje solicitado. Esta es, probablemente, la combinación más útil. Permite que un proceso envíe uno o más mensajes a varios destinos tan rápido como sea posible. Un proceso que debe recibir un mensaje antes de poder hacer alguna función útil tiene que bloquearse hasta que llegue el mensaje. Un ejemplo es el de un proceso servidor que ofrezca un servicio o un recurso a otros procesos.

• ***Envío no bloqueante, recepción no bloqueante:*** Nadie debe esperar. El *send* no bloqueante es la forma más natural para muchas tareas de programación concurrente. Por ejemplo, si se usa para solicitar una operación de salida, tal como una impresión, permite al proceso solicitante realizar la

solicitud en forma de mensaje y continuar. Un posible riesgo del *send* no bloqueante es que un error puede llevar a una situación en la que el proceso genere mensajes repetidamente. Como no hay bloqueo para hacer entrar en disciplina al pro-ceso, esos mensajes pueden consumir recursos del sistema, incluido tiempo del procesador y espacio en buffer, en detrimento de otros procesos y del sistema operativo. Además, el *send* no bloqueante carga sobre el programador el peso de determinar qué mensaje se ha recibido: Los procesos deben emplear mensajes de respuesta para acusar La recepción de un mensaje.

Para la primitiva *receive,* la versión bloqueante parece ser la más natural para muchas tareas de programación concurrente. En general, un proceso que solicita un mensaje necesitará la información esperada antes de continuar. Sin embargo, si se pierde un mensaje, un proceso receptor puede quedarse bloqueado indefinidamente. Este problema puede resolverse por medio del *receive* no bloqueante. Sin embargo, el riesgo de esta solución es que si un mensaje se envía después de que un proceso haya ejecutado el correspondiente *receive,* el mensaje se perderá. Otro método posible consiste en permitir a un proceso comprobar si hay un mensaje esperando antes de ejecutar un *receive* y en permitir a un proceso especificar más de un origen en una primitiva *receive.* Esta última técnica resulta útil si un proceso espera mensajes de mas de un origen y puede continuar si llega cualquiera de estos mensajes.

TABLA 4.4 Características de diseño de sistemas de mensajes para la comunicación y sincronización entre procesos

Sincronización	**Formato**
Send	Contenido
bloqueante	Longitud
no bloqueante	fija
Receive	variable
bloqueante	
no bloqueante	**Disciplina de Cola**
comprobación de llegada	
	FIFO
Direccionamiento	Prioridades
Directo	
envío	
recepción	
explicita	
implícita	
Indirecto	
estático	
dinámico	
propiedad	

Direccionamiento

Evidentemente, es necesario disponer de alguna forma de especificar en la primitiva *send* qué proceso va a recibir el mensaje. De forma similar, la mayoría de las implementaciones permiten a los procesos receptores indicar el origen del mensaje que se va a recibir.

Los distintos esquemas para hacer referencia a los procesos en las primitivas *sena* y *receive* se encuadran dentro de dos categorías: direccionamiento directo e indirecto. Con el **direccionamiento directo,** la primitiva *sena* incluye una identificación específica del proceso destino. La primitiva *receive* se puede gestionar de dos formas. Una posibilidad requiere que el proceso designe explícitamente un proceso emisor. Así pues, el proceso debe conocer de antemano de qué proceso espera un mensaje. Esto suele ser eficaz para

procesos concurrentes y cooperantes. En otros casos, sin embargo, es imposible especificar el proceso de origen por anticipado. Un ejemplo es un proceso servidor de impresoras, que aceptará mensajes de solicitud de impresión de cualquier otro proceso. Para tales aplicaciones, una técnica más efectiva consiste en usar direccionamiento implícito. En este caso, el parámetro *origen* de la primitiva *receive* tendrá un valor de retomo cuando se haya realizado la operación de recepción.

El otro enfoque es el **direccionamiento indirecto.** En este caso, los mensajes no se envían directamente del emisor al receptor, sino a una estructura de datos compartida formada por colas que pueden guardar los mensajes temporalmente. Estas colas se denominan generalmente *buzones (mailboxes)*. De este modo, para que dos procesos se comuniquen, uno envía mensajes al buzón apropiado y el otro los coge del buzón.

Una ventaja del direccionamiento indirecto es que se desacopla a emisor y receptor, permitiendo una mayor flexibilidad en el uso de los mensajes. La relación entre emisores y receptores puede ser uno a uno, de muchos a uno, de uno a muchos o de muchos a muchos. Una relación uno a uno permite que se establezca un enlace privado de comunicaciones entre dos procesos, lo que aísla su interacción de injerencias erróneas de otros procesos. Una relación de muchos a uno resulta útil para interacciones cliente/servidor, donde un proceso ofrece un servicio a un conjunto de procesos. En este caso, el buzón se denomina *puerto* (figura 4.25). Una relación uno a muchos permite un emisor y muchos receptores; es útil para aplicaciones en las que un mensaje o alguna información se difunda a un conjunto de procesos.

La asociación de procesos a buzones puede ser estática o dinámica. Los puertos suelen estar asociados estáticamente con algún proceso en particular, es decir, el puerto se crea y se asigna al proceso permanentemente. De forma similar, una relación uno a uno normalmente se define de forma estática y permanente. Cuando hay varios emisores, la asociación de un emisor a un buzón puede realizarse dinámicamente. Se pueden usar primitivas como *conectar* y *desconectar* con este propósito.

Una cuestión afín es la de la propiedad del buzón. En el caso de un puerto, normalmente pertenece y es creado por el proceso receptor. De este modo, cuando se destruye el proceso, también se destruirá el puerto. Para el caso general de los buzones, el sistema operativo puede ofrecer un servicio de creación de buzones. Estos buzones pueden ser considerados como propiedad del proceso creador, en cuyo caso se destruyen junto con el proceso o pueden ser considerados como propiedad del sistema operativo, en cuyo caso se necesita una orden explícita para destruir el buzón.

```
program      mutualexclusion;
const n = . . . ; (*number of processes*);
procedure P(i: integer);
var msg: message;
begin
    repeat
        receive (mutex, msg);
        < critical section >;
        send (mutex, msg);
        < remainder >
    forever
end;
begin (* main program *)
    create_mailbox (mutex);
    send (mutex, null);
    parbegin
        P(1);
        P(2);
        . . .
        P(n)
    parend
end.
```

Figure 5.26 Mutual Exclusion Using Messages

15.1.5.4.1 Problema de productor consumidor:

```
const
    capacity = ...;      {buffering capacity}
    null = ...;          {empty message}
var    i: integer;
procedure producer;
    var pmsg: message;
    begin
        while true do
            begin
                receive (mayproduce, pmsg);
                pmsg := produce;
                send (mayconsume, pmsg)
            end
    end;
procedure consumer;
    var csmg: message;
    begin
        while true do
            begin
                receive (mayconsume, cmsg);
                consume (csmg);
                send (mayproduce, null)
            end
    end;

{parent process}
begin
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for i = 1 to capacity do send (mayproduce, null);
    parbegin
        producer;
        consumers
    parend
end
```

Figure 5.27 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Messages

15.1.5.4.2 Problema de los lectores escritores:

Una solución alternativa, que da prioridad a los escritores y que se implementa mediante paso de mensajes, se muestra en la figura 4.30 y está basada en un algoritmo tomado de un trabajo de Theaker y Brookes [THEA83]. En este caso, hay un proceso controlador que tiene acceso al área de datos compartida. Los otros procesos que desean acceder a los datos envían un mensaje de solicitud al controlador, concediéndose el acceso mediante un mensaje de respuesta "OK" e indicándose la finalización del acceso con un mensaje "terminado".
El controlador dispone de tres buzones, uno para cada tipo de mensaje que debe recibir. El proceso controlador da servicio a los mensajes de solicitud de escritura antes que a las solicitudes de lectura, para dar prioridad a los escritores. Además, se respeta la exclusión mutua. Para hacerla cumplir, se emplea la variable *cont,* que se inicializa con algún número mayor que el número máximo de lectores posible. En este ejemplo, se emplea un valor de 100. Las acciones del controlador pueden resumirse de la siguiente forma: • Si *cont* > *0,* no hay escritores esperando y puede haber o no lectores activos. Servir todos los mensajes "terminado" antes de eliminar los lectores activos. A continuación, servir las solicitudes de escritura y después las de lectura. • Si *cont* = *0,* la única solicitud pendiente es de escritura. Permitir al escritor continuar y es-perar un mensaje "terminado". • Si *cont* < *0,* un escritor ha realizado una solicitud y se le ha hecho esperar para despejar to-dos los lectores activos. Por tanto, sólo deben servirse mensajes "terminado".

```
procedure readeri;
    var rmsg: message;
    begin
        repeat
            rmsg := i;
            send (readrequest, rmsg);
            receive (mboxi, rmsg);
            READUNIT;
            rmsg := i;
            send (finished, rmsg)
        forever
    end;
procedure writerj;
    var rmsg: message;
    begin
        repeat
            rmsg := i;
            send (writerequest, rmsg);
            receive (mboxj, rmsg);
            WRITEUNIT;
            rmsg := i;
            send (finished, rmsg)
        forever
    end;
```

```
procedure controller;
    begin
        repeat
            if count > 0 do begin
                if not empty (finished) then begin
                    receive (finished, msg);
                    count := count + 1
                    end
                else if not empty (writerequest) then begin
                    receive (writerequest, msg);
                    writer.id := msg.id;
                    count := count - 100
                    end
                else if not empty (readrequest) then begin
                    receive (readrequest, msg);
                    count := count - 1;
                    send (msg.id, "OK")
                    end
            end;
            if count = 0 do begin
                send (writer.id, "OK");
                receive (finished, msg);
                count := 100
            end;
            while count < 0 do begin
                receive (finished, msg);
                count := count + 1
            end
        forever
    end.
```

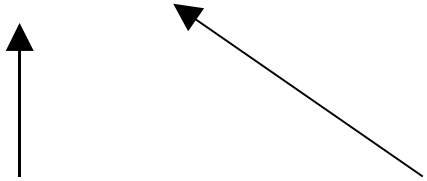
Figure 5.30 A Solution to the Readers/Writers Problem Using Message-Passing

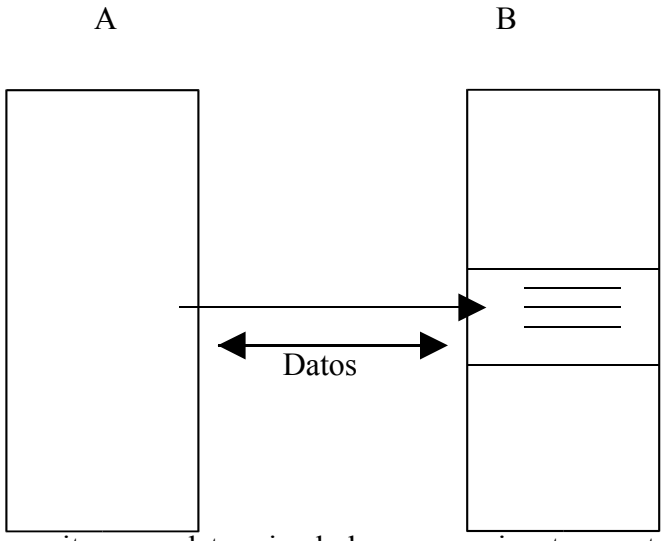
15.1.5.5 ADA:

¿Cómo maneja ADA la comunicación y sincronización de TAREAS?

Utiliza la idea de RANDEZVOUS → CITA o ENCUENTRO
Alguien solicita una cita con alguna persona, y esta los atiende según orden de llegada

Se ejecutan concurrentemente y A pide una cita a B





A solicita a B una cita en un determinado lugar para ejecutar una tarea o proceso de B, allí existe la transferencia de datos.

Para A es visto como la ejecución de una subrutina.

El procedimiento que solicito la cita con algún procedimiento queda bloqueado hasta que el otro procedimiento acepte la cita. Puede ocurrir que el proceso B este esperando una cita entonces el proceso B se quedará bloqueado hasta que algún proceso le solicite una cita.

(puede ser cualquier proceso)

Nota.-

- Una vez que el proceso B acepta una cita atiende primero a quien primero solicito la cita.
- Durante el encuentro se ejecuta un trozo de código común
- Este es un mecanismo de sincronización y transferencia de datos entre los procesos

Sintaxis de ADA para alguna de estas cosas.

```
Procedure ALGO
        task ....
        ----------
        ----------
        end

        task ....
        ----------
        ----------
        end
        begin
```

```
        ----------
        ----------
endalgo
```

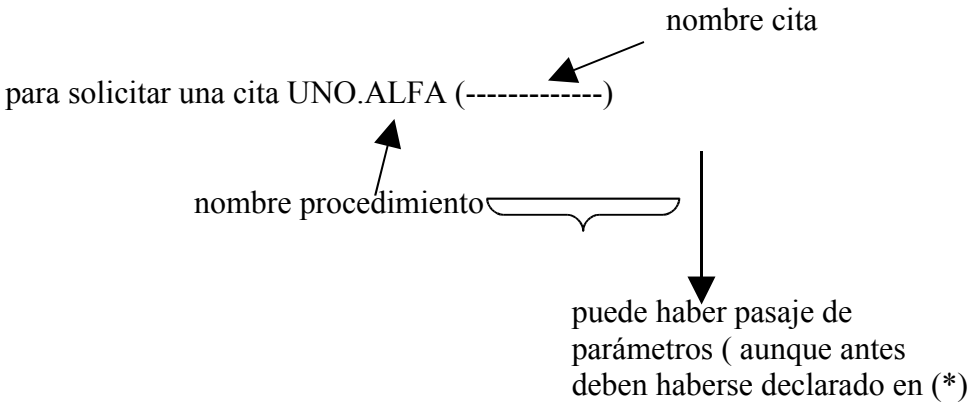
La diferencia entre nombrar un procedimiento como procedimientos o como task es que los tasks comienzan su ejecución concurrentemente luego del begin del procedure principal (ALGO).
El pasaje de parámetros a las tasks se hace através de las citas. (si hay algo entre el begin y el end esto se ejecuta concurrentemente con las tareas).

Nota.- ¡¡¡ OJO ¡!! No utilizar variables globales en ADA
NUNCA ¡!!

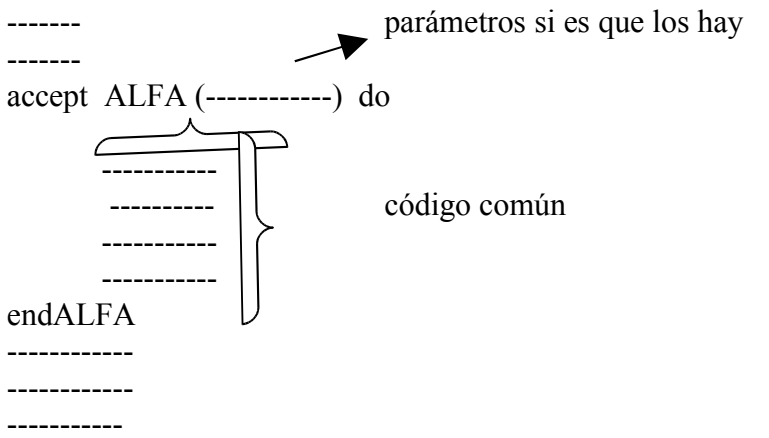
Definición de una tarea.-

parte declarativa (*)

```
task  UNO is
        entry ALFA (-);  se declaran los nombres de
        entry BETA (-);  los lugares donde acepta
end UNO                    citas (sólo se podrán
                         aceptar citas con esos
                           nombres)

task body UNO is
        ----------
        ----------
        ----------
end UNO
```



la aceptación de una cita es de la siguiente forma:



e iría en el cuerpo del procedimiento UNO

nota.- más abajo podría haber otro accept ALFA con otro código.

Para realizar únicamente una sincronización :

```
Accept beta;
```

Ej.- de pasaje de parámetros.-

```
task UNO is
        entry ALFA (A: in integer, B: out character, C:in out real);
        entry BETA
end UNO

accept ALFA (A: in integer, B: out character, C:in out real);
----------
----------
----------
----------
```

estos parámetros son los que se utilizarán en el cuerpo, además de todos aquellos que fueron declarados antes en la tarea UNO

solicitud de cita → tarea . encuentro (params)
donde encuentro es el nombre de la cita

aceptación de cita → accept encuentro (params) do

---
end encuentro

Nota.- puede ser que no haya parámetros ni cuerpo para el accept
esto implicaría, que es utilizado para la sincronización de procesos.
No utiliza variables (memoria) compartida => funciona bien en Redes.

15.1.5.5.1 Problema de ALICIA y BERNARDO

```
task PATIO is                  task      ALICIA;              task BERNARDO;
  entry PEDIR          task body ALICIA is     task body BERNARDO is
  entry DEVOVER          begin                                  .....
end                              loop                      .....
task body PATIO is        ....VIVIR                        .....
  begin                            PATIO.PEDIR            (idem ALICIA)
    loop                            paseo_perro                .....
        accept PEDIR             PATIO.DEVOLVER        .....
        accept DEVOLVER     endloop                    .....
     endloop              end ALICIA
endPATIO

begin
end;
```

Nota.- recordemos que los procedimientos definidos como tasks comienzan a ejecutarse concurrentemente luego del begin del principal.

Nota.- el accept acepta al primero de los procedimientos que hizo el pedido de cita.

Para implementar semáforos en ADA

```
task SEMAFORO is
                entry WAIT
                entry SIGNAL
end
task body SEMAFORO is
```

```
begin
        loop
                accept WAIT
                accept SIGNAL
        endloop
end
```

RANDEZVOUS → por lo menos igual de potente que los semáforos.

Para resolver el problema de que para tener 100 semáforos habría que tener 100 tareas => en ADA se define =>

task type SEMAFORO is
(idem anterior)

luego se podrían definir variables
sean A, B, C :semáforo; → ahora tengo 3 semáforos que son 3 tareas exactamente iguales pero son <> tareas)
también se podría definir un array de tareas
sea:

```
        F: array ( 1..3 ) of semáforo;
```

Ventaja → ahora puedo hacer referencia a F (i) con i: 1..3

Ej.- (Del factorial)

```
        Procedure FACTORIAL is
                Procedure PRINT ( i : in integer) is separate;
                Procedure GET ( i : out integer ) is separate;
                N , X, Y : integer;
                task type  SEMIFACT
                        entry NUMERO ( i : in integer);
                        entry RESULT ( i : out integer)
                end SEMIFACT
                task body SEMIFACT is separate
                A, B: SEMIFACT;
                Begin
                        GET (n)
                        A.NUMERO ( n )
                        B.NUMERO ( n – 1 )
                        A.RESULTADO ( X )
                        B.RESULTADO ( Y )
                        PRINT ( X * Y)
                End FACTORIAL;

                task body SEMIFACT is
                u : integer;
```

```
        begin
                accept NUMERO ( i : in integer) do
                        u := i;
                end NUMERO
                u :=    SEMI ( u )
                accept RESULTADO ( i : out integer ) do
                        i := u;
                end RESULTADO;
        end SEMIFACT;
```

15.1.5.5.2 productor – consumidor con un buffer simple:

(buffer de un solo lugar)

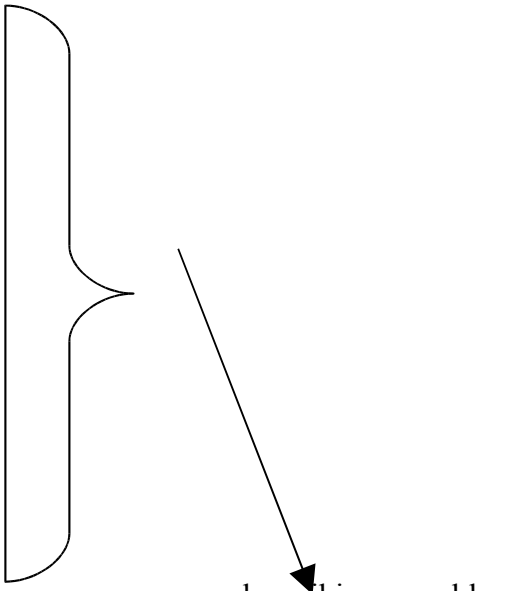
```
task  BUFF_SIMPLE is
        entry LEER ( A : out character ) ;
        entry ESCRIBIR ( A : in character );
end BUFF_SIMPLE;
task body BUFF_SIMPLE;
    x : character
        begin
                loop
                        accept ESCRIBIR (A : in character ) do
                                x  := A
                        end
                        accept LEER ( A : out character ) do
                                A := x
                        End
                Endloop
        End

task PRODUCER;
task body PRODUCER is
    cr : character
        begin
                loop
                ..... produce c1;
                BUFF_SIMPLE.ESCRIBIR (c1)
                endloop
        end
```

consumidor es análogo al productor excepto que llama a leer cuando el productor llama a escribir.

Si yo quisiera tener un buffer con más de 1 elemento podría aceptar escribir varias veces sin haber leído ninguna => se puede realizar con una nueva herramienta => SELECT

SELECT → permite seleccionar entre distintos encuentros

```
SELECT
        WHEN buff_no_lleno ( )
        accept ESCRIBIR ( ----- )
        ------------
        ------------
        end ESCRIBIR

 OR
        WHEN buff_algo ( )
        accept LEER ( ----- )
        ----------
        ----------
        ----------
        end LEER
end SELECT
```



espera por el escribir o por el leer

sale ante cualquier solicitud de leer o escribir, pero si hay llamado para ambas se elegirá una de las dos al azar.

Nota.- sería algo similar a un if dado que una vez que se acepta una de las dos se continúa con la ejecución de las líneas que vengan luego del accept hasta el OR o endSELECT

```
SELECT [  when COND1 ]
        accept ENTRADA1 (------)
                        .....
                        .....
        end ENTRADA1
        ............
        ............
  OR    [  when  COND2 ]
        accept ENTRADA2 ( ----)
                .......
                .......
        end ENTRADA2
        ...........
        ...........
 OR
        ............
```

```
        .....
        .....
ELSE
        .....
        .....
endSELECT
```

nota.-

- ⇨ entre los OR siempre debe existir un accept
- ⇨ (guarda) condición que se encuentra antes del accept (o sea solo ejecuta el accept si la guarda es verdadera)
    1. se evalúan las guardas
    2. solo se espera por aquellos accept cuyas guardas fueron verdaderas.

    Las guardas se evalúan una única vez al principio, no se están continuamente reevaluando)

    Si todas las guardas dan falso, provocaría un error, para ello está la clausula ELSE, se ejecuta cuando todas las guardas dan falsas, o cuando no hay nadie esperando por aquellos accept cuyas guardas dieron verdadero (si hay ELSE recordar que no es obligatorio) => en este 2do caso el accept no espera.

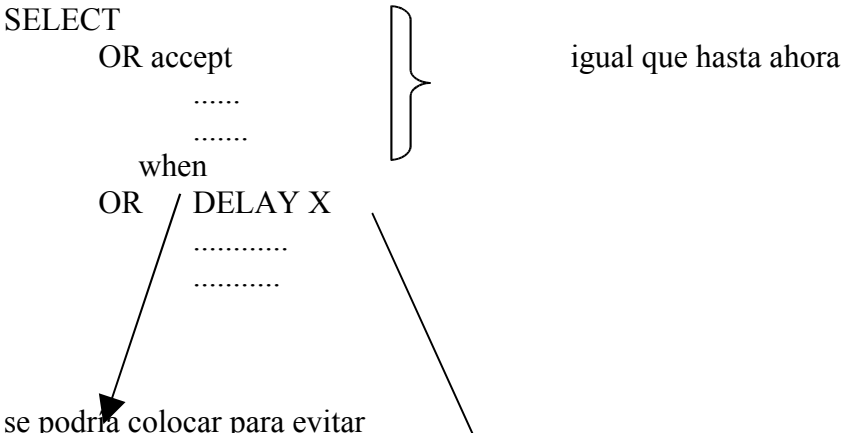
29.10

1. evalúa guardas para determinar alternativas abiertas
2. si hay alternativa abierta => se fija para esta si hay algún proceso esperando
3. si hay proc => ejecuta una de las alternativas en forma arbitraria
4. si no hay alternativa abierta o si no hay proceso en espera => ejecuta el ELSE
5. si no hay ELSE y hay alternativa abierta y nadie está solicitando => espera
6. si no hay alternativa abierta y no hay ELSE da ERROR

```
SELECT
        [ when COND1 ]
        accept ENTRADA ( ------)
                .....
                .....
                .....
        end ENTRADA
        .....
        .....
{ OR
        [ when COND2 ]
        accept .....
        .....
        .....
        .....
        ..... }
[ ELSE
        .....
        .....
```

.....]
endSELECT

existe otra opción más para el select esta sería:



se podría colocar para evitar que diera un error cuando no hay guardas abiertas (pero en general no lleva guarda)

se le indica que espere X segundos (debe haber alguna alternativa abierta, pero que nadie esta solicitando un encuentro, entonces se espera X alguna de las tareas cuyas alternativas están abiertas, durante esos X segundos se realiza el encuentro, luego de que pasan los x segundos se sale por el DELAY, en caos de no haber guardas abiertas da un error)

OJO .- No confundir con la instrucción DELAY, no es lo mismo

A diferencia del ELSE con el OR DELAY no se sale inmediatamente (el ELSE podría verse como un OR DELAY 0)

EJ- (del PRODUCTOR-CONSUMIDOR ahora utilizando un buffer finito)

```
loop
        SELECT
                when  n <  MAXIMO  =>
                        accept AGREGAR (d : in dato ) do
                                buff ( in ) := d ;
                        end AGREGAR
                        n := n + 1
                        in := ( in + 1 ) mod MAXIMO ;
        OR
                when n > 0 =>
                        accept TOMAR ( d : out dato )
                                d : = buff ( out )
                        end TOMAR
                        n := n – 1
                        out : = ( out + 1 ) mod MAXIMO ;
        end SELECT
endloop
```

15.1.5.5.3 Implementación de un semáforo no binario

```
task  SEMAFORO is
        entry INIT ( i : in integer);
                entry P
                entry V
end SEMAFORO

task body SEMAFORO is
var s : integer  // valor del semáforo
begin
        accept INIT ( i : in integer)
                s := i
        end INIT
        loop
                SELECT
                        when s > 0 =>
                                accept P ;
                                s := s – 1 ;
                        OR
                                accept V ;
                                s := s + 1 ;
                endSELECT
        endloop
endSEMAFORO
```

aquí también se puede averiguar cuantos procesos hay en espera de un accept => sería

entrada ‘ count (sólo se puede utilizar en una tarea que tenga definido un entry con ese nombre)

nombre de un entry

ej.- (podríamos en el caso de los semáforos sólo realizar un V cuando haya algún proceso esperando en un P)

```
SELECT
        when s > 0 =>
                accept P ;                      no hay procesos trancados por el P
                s := s – 1 ;
   OR
        when P ‘ count = 0 =>
                accept V ;                      hay al menos un proceso trancado por
                s := s + 1 ;                    el V
   OR
        when P ‘ count <> 0 =>
                accept V ;        // dejo continuar a quien hace el V
                accept P ;        // destrabo uno de los procesos trabados
endSELECT
```

OJO .- soluciones con ELSE podrían causar BUSY WAITING

Esta es una solución mejor de implementación de semáforos no binarios.-

Ej.- Problema que utiliza prioridades entre los procesos por lo cual utilizaremos “ count “) ;

```
task ATENDER is
        entry MAX (---)                 // MAX, MEDIA y MIN indican las
        entry MEDIA (---)     // prioridades de entradas ej. MAX es
        entry MIN (---)                 // la de mayor prioridad
end ATENDER
task body ATENDER is
begin
        loop
                SELECT
                        accept MAX (---)
                        ----------
                        ----------
                        ----------
                OR   when  MAX ‘ count = 0
                                accept MEDIA (----)
                                ----------
```

```
                    ----------
          OR  when  MAX ‘ count = 0 AND MEDIA ‘ count = 0
                    accept MIN (----)
                    ---------
                    ---------
     endSELECT
endloop
end ATENDER
```

Nota.- en ADA también se podría definir un array de entradas
Ej.- task .....;
entry PRIORIDADES (tipo) (parametros)
end tarea

la única diferencia es en la forma en la que hacemos referencia a las entradas.

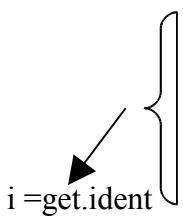
Ej.- accept prioridad (alta)
Donde tipo = num { alta, media , baja }
Ej.- complicado de invocación, array de tareas con array de entradas)

Tarea (i) . entrada (j) (.....)

15.1.5.5.4 problema de los filósofos que cenan

→ tareas → filósofos (5)
→ tenedores (5)

```
Procedure FILOSOFOS
  task type TENEDOR
          entry TOMAR;
          entry DEJAR;
  end TENEDOR;
  task body TENEDOR is
     begin
       loop
          accept TOMAR ;
          accept DEJAR;
     endloop
          .......
          .......
     end TENEDOR

var T : array ( 0 .. 4 ) of TENEDOR;
task type FILOSOFO
task body FILOSOFO is
     begin
```



```
        accept IDENT ( j : in integer)     // lo hacemos para poder
                                                    // pasar parámetros a la
                                                    // tarea
                i := j                              // se le asigna un identificador
        end IDENT
i =get.ident    izq = i
        der = ( i + 1 ) mod 5 ;
        loop
                PENSAR;
                COMEDOR . ENTRAR  // comedor sería un
                T ( izq ) . TOMAR      // semáforo no binario
                T ( der ) . TOMAR      // inicializado en 4,
                Comer;                          // implementado en ADA
                T ( der ) . DEJAR
                T ( izq ) . DEJAR
        endloop
F : array ( 0 .. 4 ) of FILOSOFO;
        begin
                for i in 0 .. 4 loop
                        F ( i ) IDENT ( i )
                endfor
        end
```

otra comunicación entra TAREAS sería RPC (Remote Procedure Call) (le doy los parámetros)
→ manda ejecutar un proceso remoto y queda bloqueado aguardando el resultado.

=> con SEND y RECEIVE sería =>

RPC

```
        SEND ( proc , param )                   → el que envía
        RECEIVE ( proc , result )

        RECEIVE ( llamador , param )            e proceso remoto
                .....
                .....
        SEND ( llamador , resultados )
```

16 MicroNucleo:

Un micronucleo es un pequeño nucleo de sistema operativo que proporciona las bases para ampliaciones modulares.
La filosofia en que se basa el micronucleo es que solo las funciones absolutamene escenciales del sistema operativo deben permanecer en el nucleo.

sistema monoliticos:
La estructura consiste en que no hay estructura; El sistema operativo se escribe como una coleccion de procedimientos, cada uno de los cuales puede invocar a cualquiera de los otros cuando necesita hacerlo. Todos los procedimientos son visibles para todos los demas [tanenbaum]

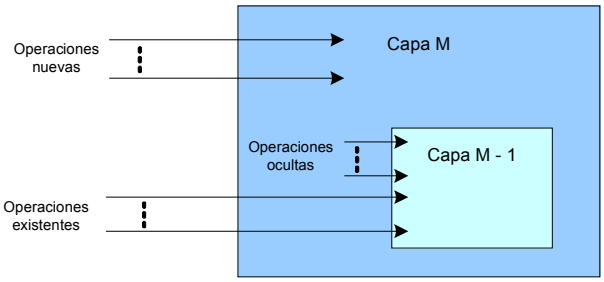
What are the main advantages of the microkernel approach to system design?
Answer: Benefits typically include the following (a) adding a new service does not require modifying the kernel, (b) it is more secure as more operations are done in user mode than in kernel mode, and (c) a simpler kernel design and functionality typically results in a more reliable operating system.

¿Cual es la diferencia entre un sistema monolítico y uno basado en microkernel?. Indique ventajas y desventajas.
Un sistema monolítico no tiene una estructura bien integrada. Todas las funcionalidades del sistema se ejecutan en modo monitor. Son difíciles de extender, mantener y depurar. Los sistemas con microkernel son sistemas estructurados, donde la mayoría de los servicios y funciones se omplementan en módulos, dejándose sólo una pequeña parte del sistema (el microkernel) que ejecute en modo monitor. En general el microkernel se limita a gestión dbásica de procesos, gestión de memoria a bajo nivle y comunicación entre procesos. Sistemas de este tipo, son mas portables y escalables que los convencionales. No obstante, debido a que los distintos componentes ejecutan en espacios de direccionamiento distintos, se presenta un overhead mayor al de los sistemas monóliticos.

17 Diseño por capas:

Si se cuenta con el apoyo apropiado de hardware, los sistemas operativos se pueden dividir en fragmentos más pequeños y adecuados que los originales como en el caso de MS-DOS y UNIX. Así el SO puede mantener un control mucho mayor sobre el computador y las aplicaciones que lo usan.

Una de las formas de modularizar, consiste en dividir el sistema en varias capas, cada una construida sobre las capas inferiores. La capa 0 es el hardware, y la n (la más alta) es la interfaz con el usuario.



Una capa es una implementación de un objeto abstracto, que es el encapsulamiento de datos y operaciones que manipulan esos datos.

La capa M consiste en estructuras de datos y rutinas que las capas de nivel superior pueden invocar. A su vez M puede invocar funciones de capas de niveles más bajos. Cada capa se implementa utilizando solo operaciones provistas por capas del nivel inferior.

Ventajas:

- Al utilizar solo operaciones implementadas en el nivel inferior, se simplifica la depuración y verificación del sistema.
- Se logra abstracción, ya que no sabemos como la capa inferior implementó la operación, sino que solo la usamos. Cada capa oculta las estructuras de datos, operaciones y hardware, de las capas superiores.

Desventajas:

- El principal problema es la definición apropiada, puesto que una capa solo puede usar las capas de nivel más bajo, y la planificación debe ser muy cuidadosa.
- El sistema por capas tiende a ser menos eficiente, ya que una rutina invoca a otra y así sucesivamente, creando un gasto extra.

What is the main advantage of the layered approach to system design?
Answer: As in all cases of modular design, designing an operating system in a modular way has several advantages. The system is easier to debug and modify because changes affect only limited sections of the system rather than touching all sections of the operating system. Information is kept only where it is needed and is accessible only within a defined and restricted area, so any bugs affecting that data must be limited to a specific module or layer.

The experimental Synthesis operating system has an assembler incorporated within the kernel. To optimize system-call performance, the kernel assembles routines within kernel space to minimize the path that the system call must take through the kernel. This approach is the antithesis of the layered approach, in which the path through the kernel is extended so that building the operating system is made easier. Discuss the pros and cons of the Synthesis approach to kernel design and to system-performance optimization.
Answer: Synthesis is impressive due to the performance it achieves through on-the-fly compilation. Unfortunately, it is difficult to debug problems within the kernel due to the fluidity of the code. Also, such compilation is system specific, making Synthesis difficult to port (a new compiler must be written for each architecture).

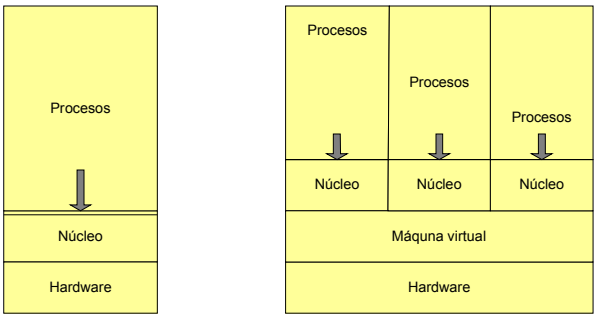
17.1.1 Maquinas Virtuales:

Conceptualmente, un sistema de computador se compone de capas. El hardware es el nivel mas bajo de todos estos sistemas.
El enfoque de maquina virtual, en cambio no proporciona funcionalidad adicional, sino que presenta una interfaz que es idéntica al hardware desnudo subyacente. Cada proceso recibe una copia (virtual) del computador subyacente.
El concepto de maquina virtual adopta el enfoque de capas y trata al nucleo del sistema operativo y al hardware como si todo fuera hardware. Incluso puede cargar otros sistemas operativos encima de esta maquina virtual.

What is the main advantage for an operating-system designer of using a virtual-machine architecture? What is the main advantage for a user?
Answer: The system is easy to debug, and security problems are easy to solve. Virtual machines also provide a good platform for operating system research since many different operating systems may run on one physical system.

Un sistema de computador se compone de capas, donde la más baja es el hardware. El núcleo que se ejecuta en el siguiente nivel, utiliza las instrucciones del hardware para crear un conjunto de llamadas al sistema, que las capas exteriores pueden usar.

Usando planificación de CPU y técnicas de memoria virtual, un sistema operativo puede crear la ilusión de que múltiples procesos se ejecutan cada uno en su propio procesador con su propia memoria. Los recursos del computador físico se comparten para crear las máquinas virtuales. Se puede usar planificación del CPU para compartirla entre los usuarios.



Un problema con la máquina virtual, tiene que ver con los sistemas de disco. Si una máquina tiene tres discos, y se quieren crear 7 máquinas virtuales, no puedo asignarle un disco a cada una. La solución es ofrecer discos virtuales a cada una de ellas, donde la suma del tamaño de los discos virtuales, es menor que el espacio físico de los discos.

17.1.2 Implementación

El concepto de máquina virtual, es simple, pero difícil de implementar. La máquina virtual en sí, solo puede ejecutarse en modo usuario, pero el software virtual, puede hacerlo en modo monitor, puesto que es el sistema operativo. En consecuencia, debemos tener un

modo usuario y monitor virtual, que se ejecutan en modo de usuario físico, por lo tanto, todas las acciones que permiten el paso de modo usuario a modo monitor en una máquina real, también deben hacerlo en una virtual.

Es posible de hacer, pero la diferencia, es el tiempo, ya que es necesario simular todas las instrucciones, y se ejecutan más lentamente.

17.1.3 Beneficios

La máquina virtual está completamente aislada de las demás, por lo tanto la protección de los recursos del sistema, es total. Los recursos no se comparten directamente, y se puede definir una red de máquinas virtuales cada una de las cuales puede enviar información a través de una red de comunicaciones virtual.

Las máquinas virtuales se están poniendo de moda para resolver los problemas de compatibilidad de sistemas. Sirve para que programas hechos para un sistema determinado puedan ser ejecutados en otro sistema no compatible.

17.1.4 Java

El compilador de Java genera *códigos de bytes* como salida, que se ejecutan en la *Máquina Virtual Java (JVM)*. Cada sistema tiene su JVM, que controla seguridad y confiabilidad del programa. Por lo tanto se logran programas seguros y transportables, que a pesar que no son tan rápidos como los compilados para hardware nativo, si son más eficientes, y tienen ventajas sobre ellos.

18 Gestion de la memoria:

En un sistema monoprogramado, la memoria principal se divide en dos partes: una parte para el sistema operativo (monitor residente, núcleo) y otra parte para el programa que se ejecuta en ese instante. En un sistema multiprogramado, la parte de "usuario" de la memoria debe subdividirse aún más para hacer sitio a varios procesos. La tarea de subdivisión la lleva a cabo dinámicamente el sistema operativo y se conoce como **gestión de memoria.**

En un sistema multiprogramado resulta vital una gestión efectiva de la memoria. Si sólo hay unos pocos procesos en memoria, entonces la mayor parte del tiempo estarán esperando a la E/S y el procesador estará desocupado. Por ello, hace falta repartir eficientemente la memoria para meter tantos procesos como sea posible.

18.1.1 Reubicación:

Cuando se emplea el esquema de particiones fijas de la figura 6.3a, se puede esperar que un proceso sea asignado siempre a la misma partición. Es decir, la partición que se selecciona cuando se carga un nuevo proceso será la misma que se emplee siempre para devolver ese proceso a memoria, tras haber sido sacado. En este caso, se puede emplear un cargador sencillo, tal como se describe en el Apéndice 6A: Cuando el proceso se carga por primera vez, todas las referencias relativas a memoria en el código se reemplazan por direcciones absolutas de memoria principal, determinadas por la dirección base del proceso cargado.
En el caso de particiones de igual tamaño y en el caso de una cola única de procesos para particiones de distinto tamaño, un proceso puede ocupar diferentes particiones a lo largo de su vida. Cuando al principio se crea la imagen de un proceso, se cargará en alguna partición de memoria principal. Posteriormente, el proceso puede ser descargado; cuando, más tarde, vuelva a ser cargado, podrá asignársele una partición distinta de la anterior. Esto mismo se cumple con particiones dinámicas. Obsérvese en las figuras 6.4c y 6.4h que el proceso 2 ocupa dos regiones de memoria distintas en las dos ocasiones en las que se le trae a memoria. Es más, cuando se usa compactación, los procesos son desplazados durante su estancia en memoria principal.
Considérese ahora un proceso en memoria que incluya instrucciones y datos. Las instrucciones contendrán referencias a memoria de los dos tipos siguientes:
• Direcciones de elementos de datos, empleadas en instrucciones de carga, almacenamiento y en algunas instrucciones aritméticas y lógicas.
• Direcciones de instrucciones, empleadas para bifurcaciones e instrucciones de llamada.
Ahora se demostrará que estas instrucciones no son fijas, sino que cambian cada vez que se intercambia o desplaza un proceso. Para resolver este problema, se realiza una distinción entre varios tipos de direcciones. Una dirección lógica es una referencia a una posición de memoria independiente de la asignación actual de datos a memoria; se debe hacer una traducción a dirección física antes de poder realizar un acceso a memoria. Una dirección relativa es un caso particular de dirección lógica, en el cual la dirección se expresa como una posición relativa a algún punto conocido, habitualmente el comienzo del programa. Una dirección física o dirección absoluta, es una posición real en memoria principal.
Los programas que emplean direcciones relativas en memoria se cargan por medio de cargadores dinámicos durante la ejecución (véase el estudio del Apéndice 6A). Esto significa que todas las referencias a memoria en el proceso cargado son relativas al origen del programa. Así pues, se necesita en el hardware un medio para traducir las direcciones relativas a direcciones físicas en memoria principal en el momento de la ejecución de la instrucción que contiene la referencia.
/ La figura 6.6 muestra la forma en que se realiza normalmente esta traducción de direcciones.
Cuando un proceso pasa a estado Ejecutando, se carga con la dirección en memoria principal del proceso un registro especial del procesador, a veces denominado registro base. Existe también un registro límite que indica la posición final del programa; estos valores deben asignarse cuando se carga el programa en memoria o cuando se carga la imagen del proceso. A lo largo de la ejecución del proceso se encuentran direcciones relativas. Estas direcciones incluyen los contenidos del registro de instrucción, las direcciones que aparecen en las instrucciones de bifurcación y llamada, y las direcciones de datos que aparecen en las instrucciones de carga y almacenamiento. Cada una de estas direcciones relativas pasa por dos etapas de manipulación en el procesador.

En primer lugar, se añade el valor del registro base a la dirección relativa para obtener una dirección absoluta. En segundo lugar, la dirección obtenida se compara con el valor del registro límite. Si la dirección está dentro de los omites, se puede proceder con la ejecución de la instrucción. En otro caso, se genera una interrupción al sistema operativo, que debe responder al error de algún modo.
El esquema de la figura 6.6 permite a los programas cargarse y descargarse de memoria a lo largo de la ejecución. También proporciona una medida de protección: Cada imagen de proceso está aislada por el contenido de los registros base y límite, y es segura contra accesos no desea-dos de otros procesos.

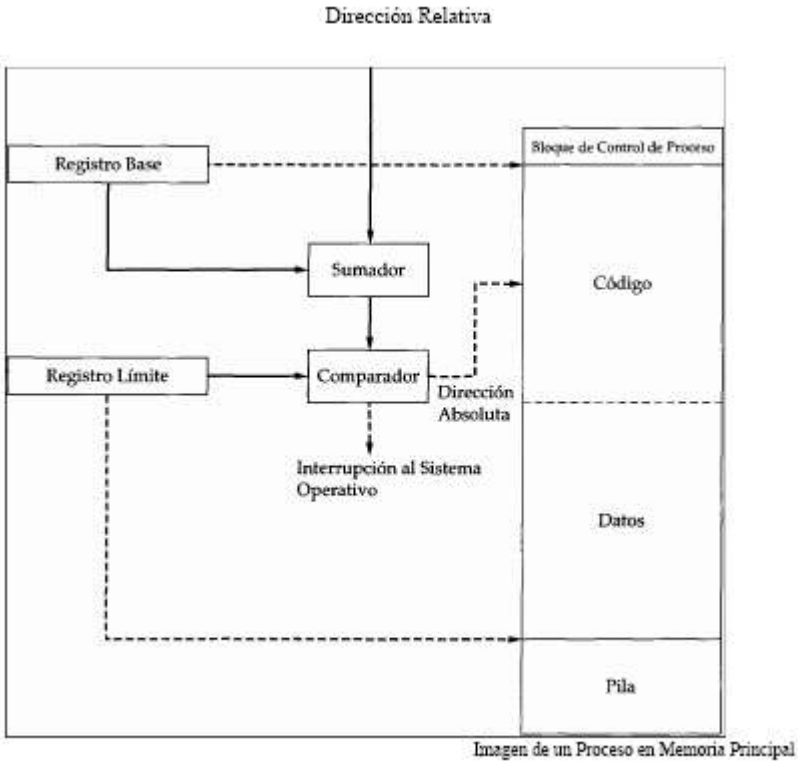


FIGURA 6.6 Soporte de hardware para la reubicación

18.1.2 CARGA Y MONTAJE:

El primer paso para la creación de un proceso activo consiste en cargar un programa en memoria principal y crear una imagen del proceso (figura 6.11). La figura 6.12 representa una situación típica para la mayoría de los sistemas. La aplicación está formada por una serie de módulos compilados o ensamblados en forma de código objeto que se montan juntos para resolver las referencias entre los módulos. Al mismo tiempo, se resuelven las referencias a rutinas de biblioteca. Las rutinas de biblioteca pueden estar incorporadas en el programa o ser referenciadas como código compartido que debe suministrar el sistema operativo en el momento de la ejecución. En este apéndice, se resumirán las características clave de los montadores y cargadores. Por claridad de presentación, se comenzará con una descripción de la tarea de carga cuando se dispone de un único módulo de programa y no se necesita montaje.

18.1.3 Carga

En la figura 6.12, el cargador sitúa el módulo de carga en la memoria principal, comenzando en la posición *x*. En la carga del programa, se deben satisfacer las necesidades de direccionamiento mostradas en la figura 6.1. En general, se pueden aplicar tres métodos:

- Carga absoluta
- Carga reubicable

• Carga dinámica en tiempo de ejecución

18.1.4 Carga absoluta

La carga absoluta necesita que el módulo de carga ocupe siempre la misma posición de memoria principal. Así pues, todas las referencias del módulo de carga para el cargador deben ser direcciones específicas o absolutas en memoria principal. Por ejemplo, si *x* en la figura 6.12 es la posición 1024, la primera palabra del módulo de carga destinado a esa región de memoria tendrá la dirección 1024.

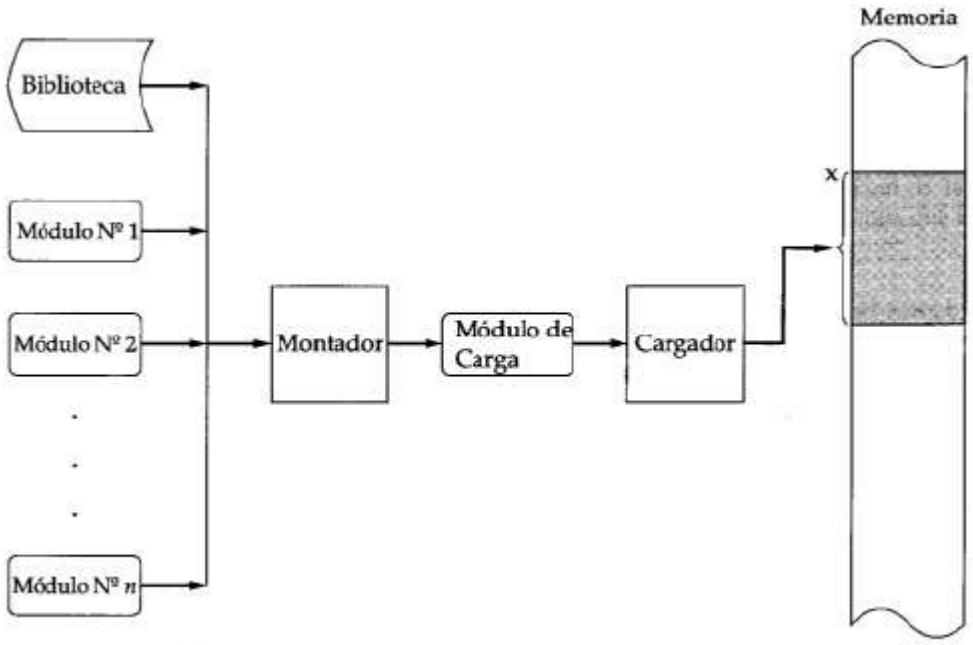


FIGURA 6.12 Proceso de carga

1

La asignación de direcciones específicas a las referencias a memoria de un programa puede ser realizada tanto por el programador como en tiempo de compilación o ensamblaje (tabla 6.2a). Con el primer método se tienen varias desventajas. En primer lugar, todos los programadores tendrán que conocer la estrategia de asignación deseada para situar los módulos en memoria principal. En segundo lugar, si se hace alguna modificación en el programa que suponga inserciones o borrados en el cuerpo del módulo, tendrán que cambiarse todas las direcciones. Por consiguiente, es preferible permitir que las referencias a memoria dentro de los programas se expresen simbólicamente y que se resuelvan en el momento de la compilación o el ensamblaje. Esto se ilustra en la figura 6.13b. Todas las referencias a una instrucción o elemento de datos se representan inicialmente por un símbolo. Cuando se prepara el módulo para la entrada a un cargador absoluto, el ensamblador o el compilador convertirán todas estas referencias a direcciones específicas (en este ejemplo, para cargar el módulo en la posición de comienzo 1024).

18.1.5 Carga reubicable

La desventaja de asociar las referencias a memoria a direcciones específicas previas a la carga es que el módulo de carga resultante sólo puede situarse en una región de memoria principal. Sin

embargo, cuando varios programas comparten la memoria principal, puede no ser conveniente decidir por adelantado en qué región de memoria debe cargarse un módulo en particular. Es mejor tomar esa decisión en el momento de la carga. Así pues, se necesita un módulo de carga que pueda ubicarse en cualquier posición de la memoria principal.
Para satisfacer este nuevo requisito, el ensamblador o el compilador no generará direcciones reales de memoria principal (direcciones absolutas) sino direcciones relativas a algún

TABLA 6.2 Enlace de Direcciones

(a) Cargador

Tiempo de Enlace	Función
Tiempo de programación	El programador especifica directamente todas las direcciones físicas reales en el propio programa.
Tiempo de compilación o ensamblaje	El programa contiene referencias simbólicas a direcciones y el compilador o el ensamblador las convierten en direcciones físicas reales.
Tiempo de carga	El compilador o ensamblador genera direcciones relativas. El cargador traduce éstas a direcciones absolutas en el instante de la carga del programa.
Tiempo de ejecución	El programa cargado conserva direcciones relativas. El hardware del procesador las convierte dinámicamente en direcciones absolutas.

(b) Montador

Tiempo de Montaje	Función
Tiempo de programación	No se permiten referencias a programas o datos externos. El programador debe situar dentro del programa el código fuente de todos los subprogramas que sean referenciados.
Tiempo de compilación o ensamblaje	El ensamblador debe ir a buscar el código fuente de cada subrutina que sea referenciada y ensamblarlas como una unidad.
Creación del módulo de carga	Todos los módulos objeto se han ensamblado usando direcciones relativas. Estos módulos se montan juntos y todas las referencias se declaran de nuevo relativas al origen del módulo de carga final.
Tiempo de carga	Las referencias externas no se resuelven hasta que el módulo de carga sea ubicado en memoria principal. En ese momento, los módulos montados y referenciados dinámicamente se añaden al módulo de carga, enviando el paquete completo a la memoria principal o virtual.
Tiempo de ejecución	Las referencias externas no se resuelven hasta que el procesador ejecute las llamadas externas. En ese momento, se interrumpe el proceso y se monta el programa llamador con el módulo que se necesita.

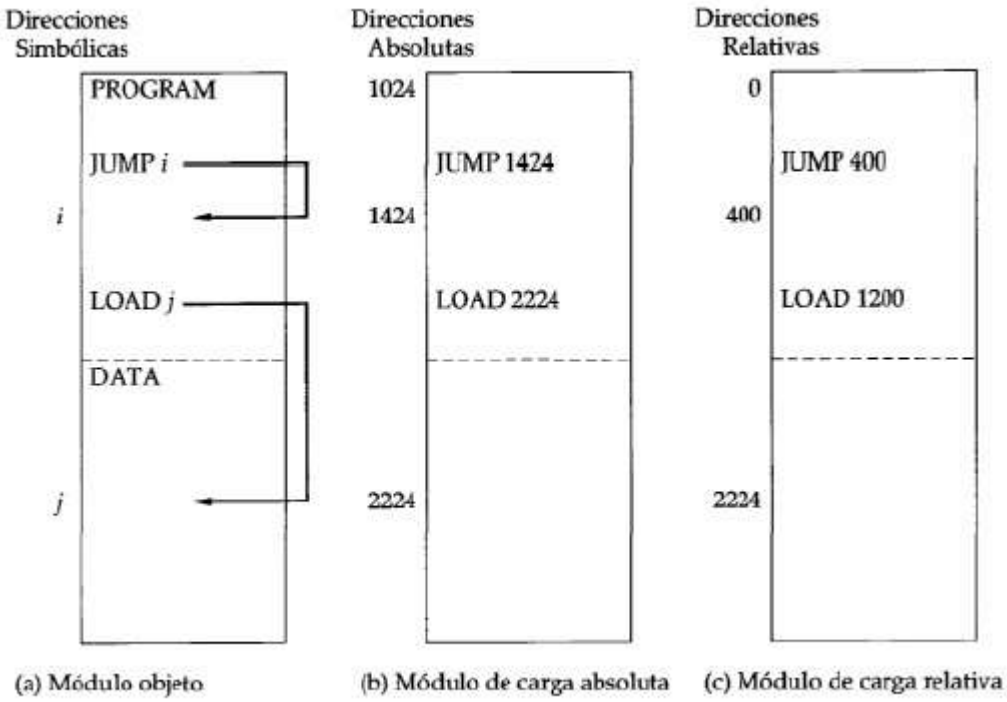


FIGURA 6.13 Módulos de carga absolutos y reubicables

punto conocido, tal como el comienzo del programa. Esta técnica se ilustra en la figura 6.13c. Al comienzo del módulo de carga se le asigna la dirección relativa O y todas las demás referencias dentro del módulo se expresan en relación al comienzo del módulo.

Con todas las referencias a memoria expresadas de forma relativa, situar los módulos en la posición deseada se convierte en una tarea sencilla para el cargador. Si el módulo va a ser cargado comenzando por la posición *x,* el cargador simplemente sumará *x* a cada referencia a memoria a medida que cargue el módulo en memoria. Para ayudar en esta tarea, el módulo de carga debe incluir información que indique al cargador dónde están las referencias a direcciones y cómo se interpretan (generalmente, de forma relativa al comienzo del programa, pero también es posible que sean relativas a algún otro punto del programa, como la posición actual). El compilador o el ensamblador prepara este conjunto de información que se conoce normalmente como *diccionario de reubicación.*

18.1.6 Carga dinámica en tiempo de ejecución

Los cargadores con reubicación son habituales y ofrecen ventajas obvias en relación con los cargadores absolutos. Sin embargo, en un entorno multiprogramado, incluso sin memoria virtual, el esquema de carga reubicable resulta inadecuado. Se ha hecho referencia a la necesidad de cargar y descargar las imágenes de procesos-de memoria principal para maximizar la utilización del procesador. Para maximizar el uso de la memoria principal, sería conveniente volver a cargar una imagen de un proceso en posiciones diferentes para diferentes instantes de tiempo. Así pues, un programa cargado puede ser expulsado a disco y vuelto a cargar en una posición distinta. Este procedimiento

resultaría imposible si las referencias a memoria hubieran estado asociadas a direcciones absolutas en el momento inicial de carga.

Una alternativa consiste en aplazar el cálculo de direcciones absolutas hasta que realmente se necesitan durante la ejecución. Con este propósito, el módulo de carga se trae a memoria con todas las referencias de forma relativa (figura 6.13c). La dirección absoluta no se calcula hasta que se ejecuta una instrucción. Para asegurar que esta función no degrada el rendimiento, debe realizarse por medio de un hardware especial del procesador, en vez de por software. Este hardware se describe en la sección 6.2.

El cálculo de direcciones dinámico proporciona una completa flexibilidad. Un programa puede cargarse en cualquier región de memoria principal. Más tarde, la ejecución del programa puede interrumpirse y el programa ser descargado de memoria principal para ser posteriormente cargado en una posición diferente.

18.1.7 Montaje

La función de un montador consiste en tomar como entrada una colección de módulos objeto y generar un módulo de carga que conste de un conjunto integrado de módulos de programa y de datos para el cargador. En cada módulo objeto pueden haber referencias a direcciones situadas en otros módulos. Cada una de estas referencias puede expresarse sólo simbólicamente en un módulo objeto no montado. El montador crea un único módulo de carga que es la concatenación de todos los módulos objeto. Cada referencia interna de un módulo debe cambiarse de dirección simbólica a una referencia a una posición dentro del módulo de carga total. Por ejemplo, el módulo A de la figura 6.14a contiene una llamada a un procedimiento del módulo B. Cuando se combinan estos módulos en el módulo de carga, esta referencia simbólica al módulo B se cambia por una referencia específica a la posición del punto de entrada de B en el módulo de carga.

18.1.8 Editor de montaje

La esencia del montaje de direcciones dependerá del tipo de módulo de carga a crear y de cuándo se produzca el montaje (tabla 6.2b). Si, como suele ser el caso, se desea un módulo de carga reubicable, el montaje se realiza generalmente de la siguiente forma. Cada módulo objeto compilado o ensamblado se crea con referencias relativas al comienzo del módulo. Todos estos módulos se unen en un único módulo de carga reubicable, junto con todas las referencias relativas al origen del módulo de carga. Este módulo puede usarse como entrada para una carga reubicable o para una carga dinámica durante la ejecución.

Los montadores que generan módulos de carga reubicables se conocen a menudo como *editores de montaje*. La figura 6.14 muestra el funcionamiento de un editor de montaje.

18.1.9 Montador dinámico

Así como en la carga, es posible aplazar algunas funciones de montaje. El término *montaje dinámico* se emplea para referirse a la práctica de retrasar el montaje de algunos módulos externos hasta después de que el módulo de carga se haya creado. Así pues, el módulo de carga contiene referencias no resueltas a otros programas. Estas referencias pueden resolverse tanto en la carga como en la ejecución.

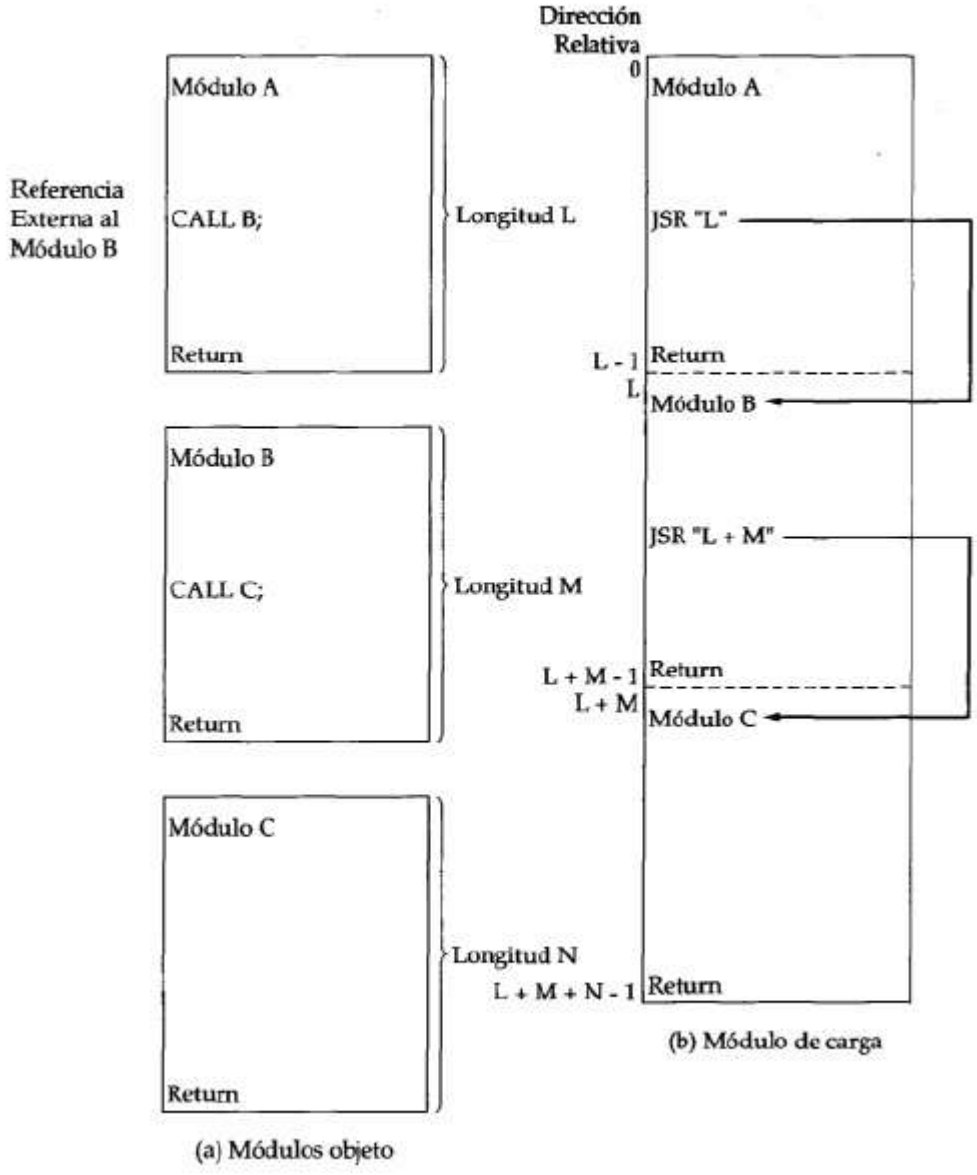


FIGURA 6.14 Funcionamiento del montaje

1

2

En el **montaje dinámico en tiempo de carga** se suceden las siguientes etapas. El módulo de carga (módulo de aplicación) se trae a memoria principal. Cualquier referencia a un módulo externo (módulo destino) hace que el cargador busque el módulo destino, lo cargue y modifique las referencias a direcciones relativas de memoria desde el comienzo

del módulo de aplicación. Existen varias ventajas en este enfoque sobre el que podría llamarse carga estática, como son las siguientes:

• Resulta más fácil incorporar cambios o actualizar versiones del módulo destino, lo que puede constituir una utilidad del sistema operativo o alguna otra rutina de propósito gene-

ral. En el montaje estático, cualquier cambio en el módulo soporte requeriría volver a montar el módulo de aplicación por completo. Esto no sólo es ineficiente, sino que puede ser imposible en determinadas circunstancias. Por ejemplo, en el campo de los computadores personales, la mayoría del software comercial se entrega en forma de módulo de carga; no se entregan las versiones fuente y objeto. • Tener el código destino en un fichero de montaje dinámico prepara el terreno para la com-partición automática de código. El sistema operativo puede darse cuenta que más de una aplicación está empleando el mismo código destino, puesto que habrá cargado y montado dicho código. Esta información puede usarse para cargar una única copia del código destino y montarla en ambas aplicaciones, en vez de tener que cargar una copia para cada aplicación. • Resulta más fácil para los productores independientes de software ampliar la funcionalidad de un sistema operativo muy empleado, como puede ser OS/2. Un productor de software puede proponer una nueva función que sea útil para varias aplicaciones y empaquetarla como un módulo de montaje dinámico. En el **montaje dinámico en tiempo de ejecución,** parte del montaje se pospone hasta el momento de la ejecución. Las referencias externas a los módulos destino permanecen en el programa cargado. Cuando se realiza una llamada a un módulo ausente, el sistema operativo localiza el módulo, lo carga y lo monta en el módulo llamador. Se ha visto anteriormente que la carga dinámica permite desplazar un módulo de carga completamente; sin embargo, la estructura del módulo es estática y permanece sin cambios a lo largo de la ejecución del proceso y de una ejecución a la siguiente. No obstante, en algunos casos, no es posible determinar antes de la ejecución qué módulos objeto harán falta. Esta situación es la normal en las aplicaciones de proceso de transacciones, como el sistema de reservas de una compañía aérea o una aplicación bancaria. La naturaleza de la transacción dictamina qué módulos de programa se necesitan y estos se cargan de la forma apropiada y se montan con el programa principal. La ventaja de emplear un montador dinámico es que no hace falta reservar memoria para las unidades de programa a menos que se haga referencia a las mismas. Esta capacidad se emplea como soporte de los sistemas de segmentación. Es posible un refinamiento adicional: Una aplicación no tiene por qué saber los nombres de todos los módulos o puntos de entrada a los que puede llamar. Por ejemplo, un programa de gráficos puede estar escrito para funcionar con una serie de trazadores gráficos, cada uno de los cuales es conducido por un paquete manejador distinto. La aplicación puede conocer el nombre del trazador instalado actualmente en el sistema a través de otro proceso o buscándolo en un archivo de configuración, lo que permite al usuario de la aplicación instalar un nuevo trazador que ni siquiera existía cuando se escribió la aplicación.

[tan-arq]
El enlazador fusiona los espacios de direcciones de los modulos objeto en un solo espacio de direcciones lineal siguiendo estos pasos:
1- Construccion de una tabla con todos los modulos objeto y sus longitudes.
2- Con base en esta tabla, se asigna una direccion de inicio a cada modulo objeto.
3- Se buscan las intrucciones que hacen referencia a la memoria y se suma a cada una de ellas una constante de reubicacion
igual a la direccion de inicio de su modulo.
4- Se buscan todas las instrucciunes que hacen referencia a otros procedimientos y se inserta la direccion del procedimineto en cuestion.

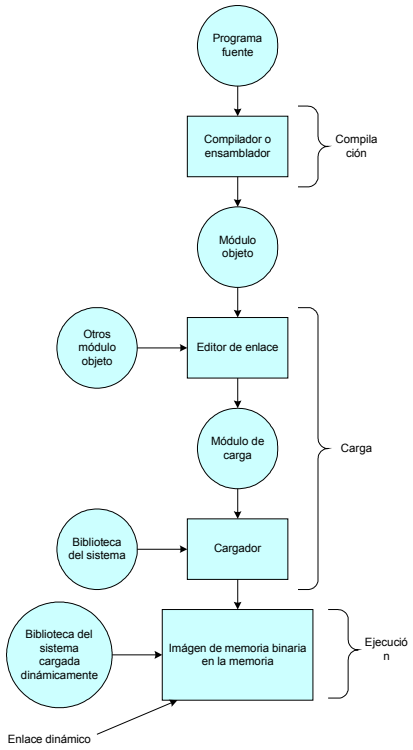
Casi todos los enlazadores requieren dos pasadas.Durante la primera pasada el enlazador lee todos los modulos objeto y construye una tabla
de nombres y longitudes de modulo, y una tabla de simbolos global que consiste en todos los puntos de ingreso y referencias externas.
Durante la segunda pasada se leen los mudulos objeto, se reubican, y se enlazan uno por uno.

cualquier cosa mira el tanenbaum (el de arq1) cuarta edicion pag 508.

[silb]

La memoria es vital para el funcionamiento del sistema. Es una gran matriz de palabras o bytes, cada uno con su propia dirección. La unidad de memoria, solo ve una corriente de direcciones que le llegan, pero no sabe como se generan, ni para que son.

18.1.10 Vinculación de direcciones



Un programa reside en el disco como un archivo binario ejecutable, y para que se ejecute es preciso traerlo a la memoria y colocarlo dentro de un proceso.

Dependiendo de la gestión de memoria, el proceso podría transferirse entre el disco y la memoria durante su ejecución, y la colección de estos procesos que esperan ser transferidos a memoria para ser ejecutados es conocida como *cola de entrada*.

Normalmente se selecciona un proceso de la cola de entrada y se carga en memoria, se ejecuta (pudiendo acceder a instrucciones y datos en memoria), y luego termina, liberando el espacio de memoria que ocupaba (se marca como disponible).

La mayor parte de los sistemas, permiten a un proceso de usuario residir en cualquier parte de la memoria física. Este proceso va a pasar por varias etapas antes de ejecutarse, y las direcciones de memoria, podrían representarse de diferentes maneras en dichas etapas.

La vinculación se puede ejecutar en cualquier etapa del camino:

- **Compilación:** Si en el momento de la compilación se sabe en que parte de la memoria va a residir el proceso, es posible generar código *absoluto*. El código empezará en la posición indicada, pero si la posición de inicio cambia, es necesario recompilar el código.
- **Carga:** Si al compilar, no se sabe en que parte de la memoria va a residir, el compilador debe generar código *reubicable*. La vinculación final se posterga hasta el momento de la carga. Si la dirección de inicio cambia, basta con volver a cargar el código.
- **Ejecución:** Si durante la ejecución, el proceso puede cambiar de un segmento de memoria a otro, la vinculación deberá postergarse hasta el momento de la ejecución. Para esto se requiere hardware especial.

18.1.11Carga dinámica

El tamaño de un proceso está limitado al tamaño de la memoria física, ya que todos los datos del proceso deben estar en la memoria para que el proceso se ejecute.

La *carga dinámica* aprovecha mejor el espacio de memoria ya que las rutinas no se cargan hasta que son invocadas, y se mantienen en el disco en formato de carga reubicable, por lo tanto, una rutina que no se utiliza, nunca se carga.

La carga dinámica no requiere soporte especial por parte del sistema operativo.

18.1.12Enlace dinámico

Si no se cuenta con el enlace dinámico, todos los programas del sistema requerirán la inclusión de una copia de las bibliotecas en su imagen ejecutable. Esto desperdicia disco y memoria principal.

Con enlace dinámico, se incluye un *fragmento* (*stub*) en la imagen por cada referencia a una biblioteca, el cual indica como localizar la biblioteca necesaria. Cuando el fragmento se ejecuta, verifica si la rutina está en memoria, y si no está, la carga, logrando una sola copia del código de la biblioteca para todos los procesos.

Si una biblioteca se sustituye por una nueva versión, todos los programas usan la nueva versión. Sin enlace dinámico, se debería volver a enlazar todos los programas con la nueva biblioteca.

Cuando hay distintas versiones de bibliotecas se los conoce como “bibliotecas compartidas”, y cada fragmento, debe guardar información tanto de la biblioteca a cargar, como de la versión.

El enlace dinámico requiere ayuda del SO, ya que para saber si una rutina ya está en memoria, debo comprobar si está en el espacio de memoria de otro proceso, y eso lo administra el SO.

[sol examenes viejos]

Reubicacion estatica:

Momento de escritura: En este caso, el programa ya contiene direcciones absolutas de memoria, para que funcione debe ser ubicado siempre en la misma dirección física de memoria.
Momento de compilación: Si utilizamos direcciones relativas, todas las direcciones internas del programa estarán referidas suponiendo que el modulo se encuentra a partir de la dirección cero de la
memoria.
En el momento de compilación se resuelven las referencias internas del modulo y las referencias externas son almacenadas en una tabla (en general al comienzo del modulo objeto) para que el linker la tome como entrada.
El linker toma como entrada la salida del compilador (tablas de referencias externas y los módulos objeto) y los une resolviendo todas las referencias externas en base a una dirección de comienzo del
programa igual a cero. El linker genera un programa y una tabla de símbolos a reubicar en el momento de carga (loading).
El loader toma el programa, la tabla de símbolos y una dirección real de memoria (provista por el Sistema Operativo) llamémosle a esta dirección RR y copia el programa a partir de la dirección RR.
Antes de comenzar a ejecutar se deben resolver las direcciones de los símbolos, o sea sumarle al contenido de cada uno el valor RR.
Una vez que se resolvieron estas direcciones el programa esta listo para ejecutar.
Reubicacion dinamica:
En este caso se traduce la dirección de un símbolo a una dirección física en tiempo de referencia.
Esta reubicación es hecha por hardware por razones de eficiencia y es transparente para el usuario.
Para este caso se utiliza una función que le llamaremos NL_map que dada una dirección de memoria virtual devuelve una dirección física de memoria.
En este caso los programas pueden ser reubicados en tiempo de ejecución y para ello lo único que se debe cambiar es la función NL_map. En este caso no es necesario cargar y linkeditar los programas antes de la ejecución sino que pueden ser cargados dinámicamente cuando se referencian.

a) Indicar que motivos existen para el uso de Linkers y Cargadores Reubicables.
b) Describir ambos procesos.
c) Mencionar en que ambientes son aplicables.
Solución 2
a) Un Linker permite unir el programa en código máquina que constituye la salida de un compilador, con otros programas también en código máquina que existan previamente (por ejemplo en bibliotecas de rutinas) y que pueden haber sido obtenidos de compilar rutinas en lenguajes distintos. Así se obtiene un solo modulo con posibilidad de ser ejecutado. Esto reduce el trabajo de programación y permite reusar lo ya hecho.
Un Cargador reubicable permite que un mismo programa (que tiene que haber sido compilado y linkeditado de forma apropiada para este tipo de cargadores) pueda ser ubicado en distintos lugares de la memoria (real o virtual, según sea un sistema de memoria virtual o no). De esta forma puede

adaptarse la carga a las necesidades y posibilidades actuales del sistema (como ser presencia de otros módulos en la memoria, o ubicación de las particiones de memoria en un sistema de estas características) lo que redunda en un mejor aprovechamiento de recursos.
b) Un Linker se basa en la información dejada por el compilador sobre las etiquetas sin resolver (referencias a nombres no definidos en el propio programa) que hallan sido usadas dentro del fuente.
Esta información se organiza en tablas que se incluyen junto con el código maquina, de esta forma pueden ubicarse los lugares en donde el compilador no logró resolver que dirección correspondía a la etiqueta y cual fue la etiqueta usada. Por cada referencia externa el Linker intenta encontrar un módulo que contenga la definición de esa etiqueta en base a los parámetros que se le hallan pasado y a las definiciones del ambiente (caminos por defecto y otros). Por cada etiqueta que logra resolver, agrega el código que la define al código que dio el compilador y en cada lugar en que esta etiqueta halla sido usada la sustituye por la dirección de su definición. Esta dirección es siempre relativa al comienzo del módulo que forman los códigos unidos, es decir un desplazamiento. Si una etiqueta no puede ser resuelta, esto es, no pudo encontrarse un módulo que la defina, se produce un error y la operación es abortada. La actividad de juntar módulos también implica reubicar símbolos y por ende corregir todas las referencias en el código a símbolos reubicables, que es lo que le exige mayor esfuerzo al linkeditor. La estructura del código objeto debe permitir ubicar los símbolos potencialmente
reubicables como consecuencia del colapso de segmentos, con las mismas características, de módulos diferentes.
Un cargador reubicable se basa en la información dejada por el compilador o el linker, la cual le permite encontrar la ubicación dentro del código de las direcciones que deben ser alteradas durante la carga para adaptarlas a la dirección real en que el programa es ubicado. Como estas direcciones son en realidad desplazamientos respecto al principio del modulo (ver descripción del Linker) el cargador le suma a cada una la dirección real de carga del código en que se encuentran, obteniendo así la dirección real donde se encuentran las definiciones de las etiquetas usadas.
c) En un ambiente donde pueda hacerse la reubicación de direcciones de memoria en momento de ejecución (al ser referenciadas), por ejemplo en un ambiente donde el mapeo este implementado por hardware.

Que es el diccionario de reubicación?
resp:
Se encuentra en el modulo objeto . Es preciso sumar una constante de reubicacion a las instrucciones que contienen direcciones de memoria. Puesto que el enlazador no tiene forma de saber por inspeccion cuales de las palabras de datos contienen instrucciones de maquina y cuales contienen constantes, esta tabla proporciona informacion acerca de cuales direcciones deben reubicarse

18.1.13Clase práctica 7

Práctico 7 - Ejercicio 1

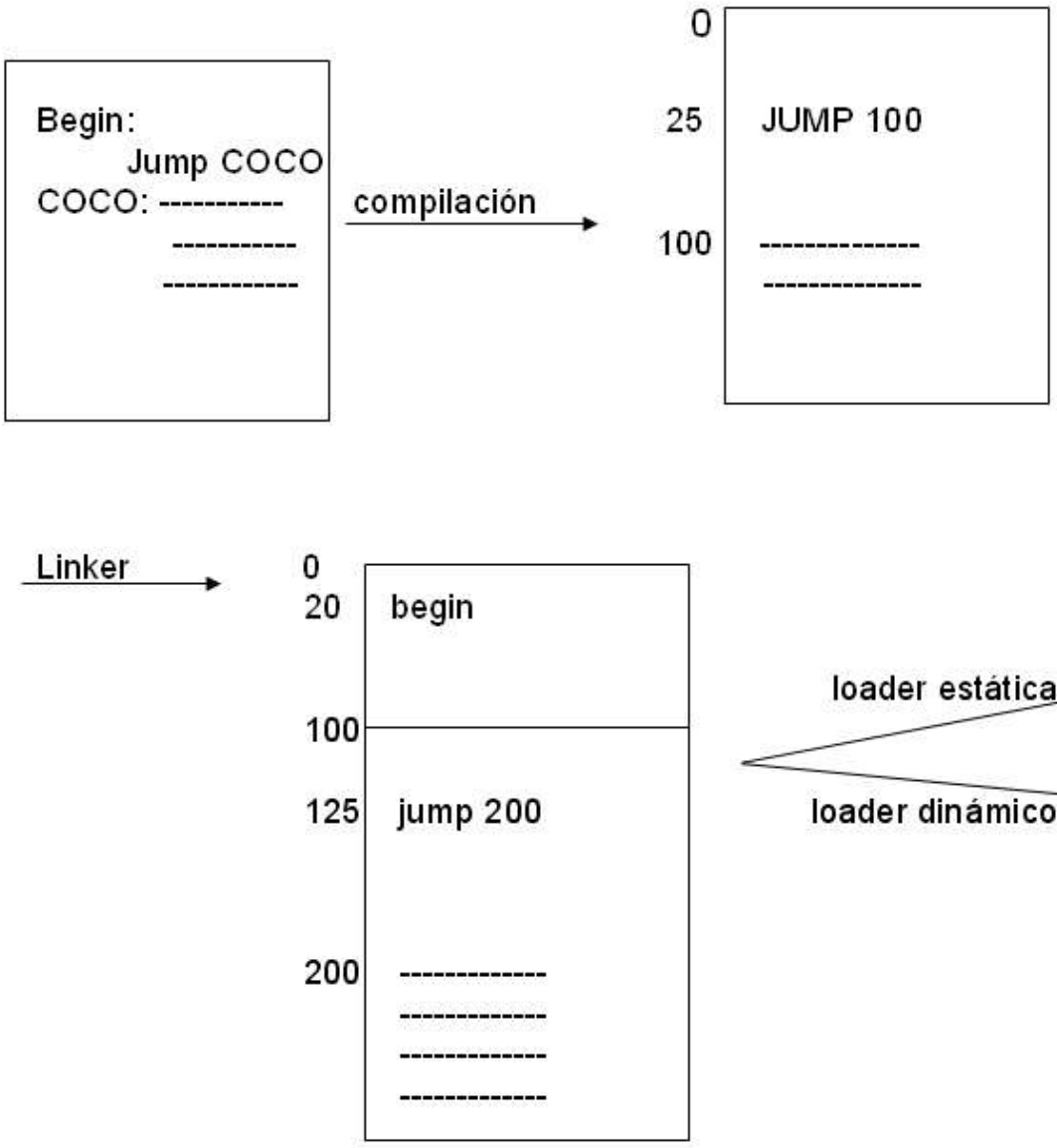
Construir un cargador de módulos que realice reubicación estática. El archivo que contiene el modulo a ser cargado tiene tres tipos de registro:
`datos_globales`: Contiene la dirección de comienzo de ejecución **relativa** al comienzo del módulo.
`codigo`: cada uno contiene el largo de esta sección del código y una sección del código.
`diccionario`: cada uno contiene la dirección de una palabra que contiene

una dirección a reubicar.
Y esta organizado de la siguiente forma:
1 registro de tipo `datos_globales`.
1 a *n* registros de tipo `codigo`.
0 a *m* registros de tipo `diccionario`.
El cargador recibirá además la dirección de carga del módulo como parámetro y deberá cargarlo en memoria a partir de esa dirección. Todas las direcciones apuntadas por el diccionario deben ser reubicadas, teniendo en cuenta que estas contienen direcciones relativas al comienzo del módulo.

```
================================================================
procedure loader(dir_carga: direccion, modulo: archivo);
var dir_act, dir_ejec : direccion;
begin
dir_ejec := datos globales; {primer registro del archivo}
dir_act := dir_carga;
para cada registro tipo codigo regCod
{carga el codigo en la direccion de memoria dir_act}
cargar(dir_act, codReg.codigo);
dir_act := dir_act + codReg.largo;
fin para;
para cada registro tipo diccionario regDic
ir a la dirección regDic.direccion + dir_carga;
sustituir el valor por (valor + dir_carga)
fin para;
ejecutar desde (dir_carga + dir_ejec) {JUMP a esa dir}
end;
================================================================
```

18.1.14Soluciones practico viejo:

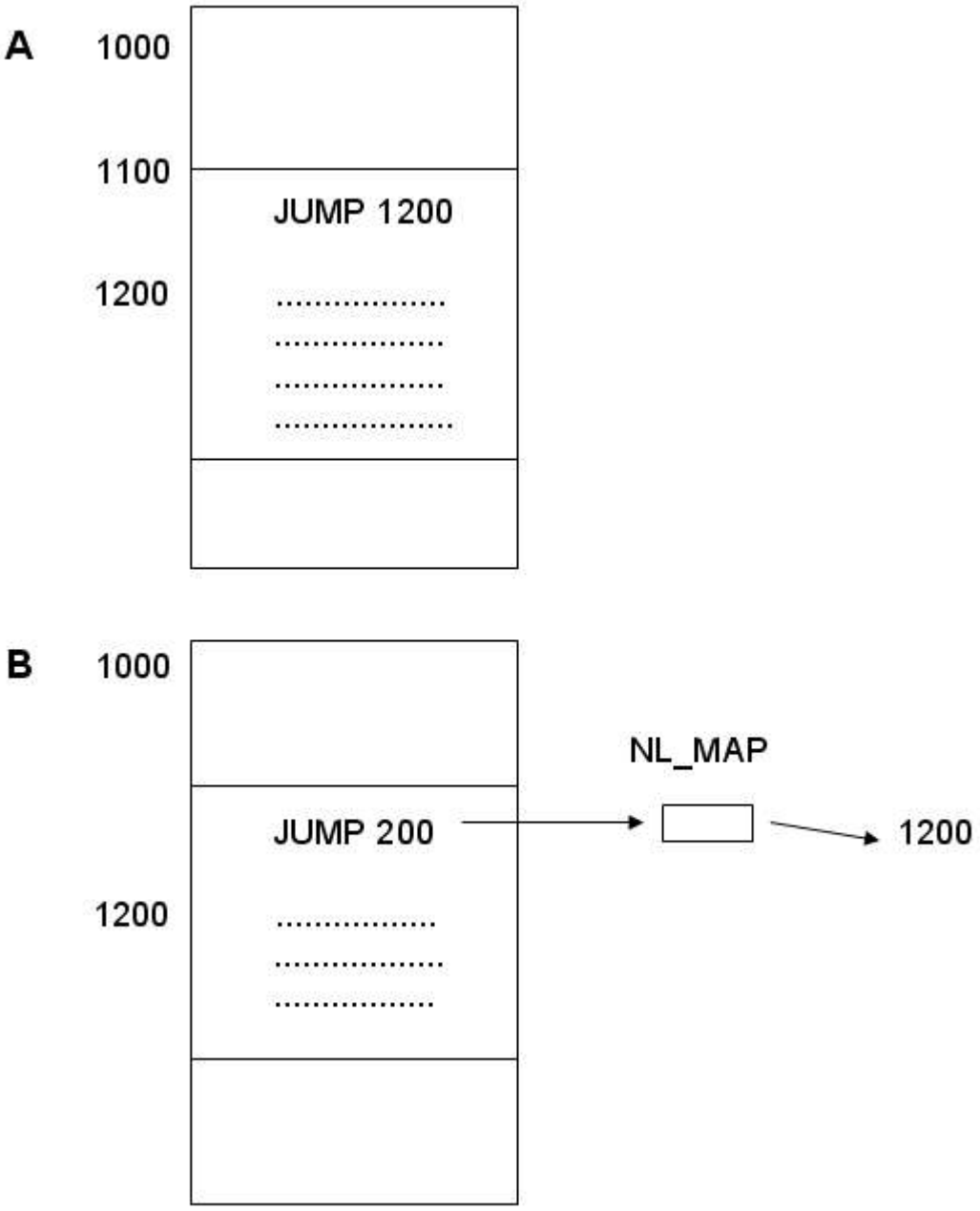
Ejercicio 8


Begin:
Jump COCO
COCO: ----------

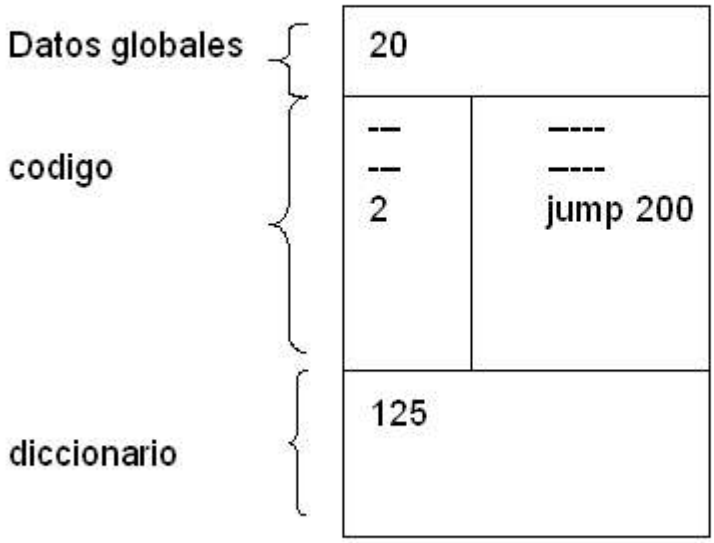
----
compilación
0
25
JUMP 100
100

Linker
0
20
begin
100
125
jump 200
200

----
loader estática A
loader dinámico B


A
1000
1100
JUMP 1200
1200
.....
.....
.....
.....
B
1000
NL_MAP
JUMP 200
1200
1200
.....
.....
.....

Recibe: y dir de carga=1000 en el ejemplo.




```
Procedure LOADER ( dir_carga: dir , modulo : archivo);
var aux : dir;
begin
      dir_ejec := datos_globales;
      aux := dir_carga;
      para cada registro tipo codigo
            CARGAR ( aux , CODIGO . codigo);
            aux := aux + CODIGO . largo;
      end;
      para cada registro tipo diccionario
 M [ dir_carga + DICCIONARIO ] := M [ dir_carga + DICCIONARIO ]
+dir_carga;
jump ( dir_carga + dir_ejec);
end;

Public                extern
           Aca , Y          Aca , X
           Modulo 1         modulo 2
           ------------           ------------
           ------------           ------------
           ------------           ------------
     Y → Aca: -----------           jump Aca ← X
```

18.1.15 ejercicio 9

```
     codigo = record diccionario : array [ 1 .. # mod * n ] of
integer;
                     datos_globales : integer;
                     array [ 1 .. maxlineas] of lineas_de_codigo;
          end

     public = array [ 1 .. #mod * M] of record begin
                                      simbolo: string;
                                       direccion: integer;
                                 end
extern = array [ 1 .. #mod * N ] of record begin
                                 simbolo : string
                                 direccion : integer;
                                 end;

procedure LINKER ( archivo : in archivo_de_modulos, A : out
codigo)
var public : public;
       extern : extern;
begin
     desplazamiento := 0; indice_e := 1 ; indice_p:= 1 ,
indice_c:= 1
     para cada módulo m del archivo_de_modulos hacer
          saco_header ( )
          para cada registro extern e de m hacer
               extern [ indice_e ], símbolo := e . símbolo;
               extern [ indice_e ] . dirección := e.dirección +
desplazamiento;
               indice_e ++ ;
          end_para;
          para cada registro public p de m hacer
               public [ indice_p] . simbolo := p . símbolo;
               public [ indice_p] . direccion := p.direccion +
desplazamiento;
               indice_p ++;
          endpara;
          para cada registro de código c de m hacer
               codigo [ indice_c ] := c . código
               desplazamiento := desplazamiento + c . largo;
               indice_c ++;
          end_para;
     end_para;
     for i := 1 to indice_e do begin
          j := 1
```

```
           encontre := false;
           while ( j < indice_p ) and not encontre do
                if ( public [ j ] . simbolo = extern [ i ] .
simbolo ) then
                     encontre := true
                else    j := j + 1;
           endwhile;
           if encontre then
             A.codigo [ extern [ i ] . direccion ] . operando :=
public
 [ j ] . dirección;
             A.diccionario [ indice_d] := extern [ i ].dirección
             indice_d++ ;
           else
              ERROR;
      endfor;
      SACO_REG_FINAL  ( archivo_de_modulos, final );
      j := 0;
           encontre := false;
      while ( j < indice_p ) and not encontre do
           if ( public [ j ] .simbolo = final . simbolo_final )
then
                encontre := true;
           else    j ++;
      endwhile;
      if encontre then
           A . datos_globales := public [ j ] . dirección;
endLinker;
```

18.1.16Examen Febrero de 2002

Ejercicio 1

Se desea construir un `linker` que deberá leer un archivo que contiene uno o más módulos, cada uno formado de la siguiente manera:

Tipo de registro Cantidad Contenido

`header` 1 nombre de módulo
`extern` 0 a *n* un símbolo y una dirección
`public` 0 a *m* un símbolo y una dirección
`codigo` 1 a *l* un largo y un código
`final` 1 un símbolo

Al final del archivo hay un registro con el siguiente formato:
`final_archivo` 1 un símbolo
Cada registro `extern` apunta a una palabra que deberá contener la dirección de un símbolo definido en otro módulo como `public`.
Cada registro `public` define la dirección de un símbolo, relativa al comienzo del módulo.
El registro de tipo `final_archivo` indica que la ejecución de este programa debe comenzar en
la dirección de carga de este símbolo.
Se pide:

- Implementar dicho linker.

```
================================================================
Ejemplo de modulos :
PUBLIC                EXTERN
----------            ----------
AQUI,y                AQUI,x
Modulo 1              Modulo 2
-----                 ------
-----                 ------
-----                 ------
y-> AQUI:  -----      JMP AQUI <-x
-----                 ------
```

En tabla Public está la dirección del símbolo, en la extern dice en que lugar hay un hueco que hay que sustituir por la dirección del símbolo (o sea en la dirección x hay que poner y).
Suponemos que la salida del linker va a un cargador dinámico, con direccionamiento por hardware
Entonces el linker será:
Recorre todos los registros Extern y forma una tabla con ellos.
Recorre todos los registros Public de los módulos y forma una tabla con ellos.
Forma un archivo de código con todos los registros de código.
Recorre la tabla Extern formada, busca en la tabla Public la dirección que corresponde a cada símbolo y sustituye en el archivo de código.

```
procedure LINKER (entrada: arch_linker, var salida: arch_load);
type simb : record
     simbolo : char;
     direccion : tipo_dir;
     sig : simbolo;
end;

var extern,public : ^simb;
dir_act : tipo_dir;
begin
dir_act := 0;
para cada modulo de entrada, mod_ent
para cada registro Extern de mod_ent, reg_ext
Agregar(extern, reg_ext.simbolo,
reg_ext.direccion + dir_inicial)
{agrega a la lista extern un registro con los valores
indicados}
para cada registro Public de mod_ent, reg_pub
Agregar(Public,reg_pub.simbolo,reg_pub.direccion + dir_inicial)
{agrega a la lista public un registro con los valores
indicados}
para cada registro Codigo de mod_end, reg_cod
```

```
Agregar(Salida,reg_cod.codigo)
{agrega al archivo de salida el codigo}
dir_act := dir_act + reg_cod.largo
{acumulo para proxima dir_act del siguiente modulo}
fin para;
para cada elemento de la lista Extern, nodo_ext
buscar en lista Public el símbolo
si no existe => error
sino=>ir a la dirección apuntada
reemplazar por la dirección de lista Public encontrada
end; {fin LINKER}
PREGUNTAR QUE PASA CON EL REGISTRO FINAL : PARA CADA MODULO NO ME
IMPORTA
AHORA,QUIERO EL PRIMERO SOLAMENTE.
================================================================
```

18.1.17Administración de memoria con mapas de bits:

Con un mapa de bits, la memoria se divide en unidades de asignación, tal vez sólo de unas cuantas palabras o quizá de varios kilobytes. A cada unidad de asignación corresponde un bit del mapa de bits, que es 0 si la unidad está libre y 1 si está ocupada (o viceversa).

El tamaño de la unidad de asignación es una cuestión de diseño importante. Cuanto menor sea la unidad de asignación, mayor será el mapa de bits.

Un mapa de bits ofrece un método sencillo para contabilizar las palabras en una cantidad fija de memoria, porque el tamaño del mapa de bits depende sólo del tamaño de la memoria y del tamaño de la unidad de asignación. El problema principal que presenta es que una vez que se ha decidido traer a la memoria un proceso de k unidades, el administrador de memoria debe buscar en el mapa de bits una serie de k bits en O consecutivos. La búsqueda de series de una longitud dada en un mapa de bits es una operación lenta (porque la serie puede cruzar fronteras de palabra en el mapa); éste es un argumento en contra de los mapas de bits.

18.1.18Administración de memoria con listas enlazadas

Otra forma de contabilizar la memoria es mantener una lista enlazada de segmentos de memoria libres y asignados, donde un segmento es un proceso o bien un agujero entre dos procesos.
Cada entrada de la lista especifica un agujero (H) o un proceso (P), la dirección en la que principia, la longitud y un apuntador a la siguiente entrada.
En este ejemplo, la lista de segmentos se mantiene ordenada por dirección. Este ordenamiento tiene la ventaja de que cuando un proceso termina o es intercambiado a disco, es fácil actualizar la lista.

Si los procesos y agujeros se mantienen en una lista ordenada por dirección, se pueden usar varios algoritmos para asignar memoria a un proceso recién creado o traído a la memoria.

18.1.19Primer ajuste:

El administrador de memoria examina la lista de segmentos hasta encontrar un agujero con el tamaño suficiente. A continuación, el agujero se divide en dos fragmentos, uno para el proceso y otro para la memoria desocupada, excepto en el poco probable caso de que el ajuste sea exacto.

18.1.20Siguiente ajuste:

Este algoritmo funciona igual que el de primer ajuste, excepto que toma nota de dónde está cada vez que encuentra un agujero apropiado. La siguiente vez que se invoque, el algoritmo comenzará a buscar en la lista a partir del lugar donde se quedó la última vez, en lugar de comenzar por el principio, como hace el primer ajuste.

18.1.21Mejor ajuste:

Examina toda la lista y toma el agujero más pequeño que es adecuado.
Lo que resulta sorprendente es que también desperdicia más memoria que el primer ajuste o el siguiente ajuste porque tiende a llenar la memoria de pequeños agujeros inútiles.

18.1.22Peor ajuste:

Tomar siempre el agujero más grande disponible, de modo que el agujero sobrante tenga un tamaño suficiente para ser útil.

Los cuatro algoritmos pueden agilizarse manteniendo listas separadas de procesos y agujeros. De este modo, los algoritmos dedican toda su energía a inspeccionar agujeros, no procesos. El inevitable precio que se paga por esta agilización de la asignación es la complejidad adicional y la lentitud del proceso de liberar memoria, ya que es preciso quitar un segmento liberado de la lista de procesos e insertarlo en la lista de agujeros.

En lugar de tener un conjunto aparte de estructuras de datos para mantener la lista de agujeros, como se hace en la Fig. 4-5(c), se pueden usar los agujeros mismos. La primera palabra de cada agujero podría ser el tamaño del agujero, y la segunda, un apuntador a la siguiente entrada. Los nodos de la lista de la Fig. 4-5(c), que requieren tres palabras y un bit (P/H), ya no son necesarios.

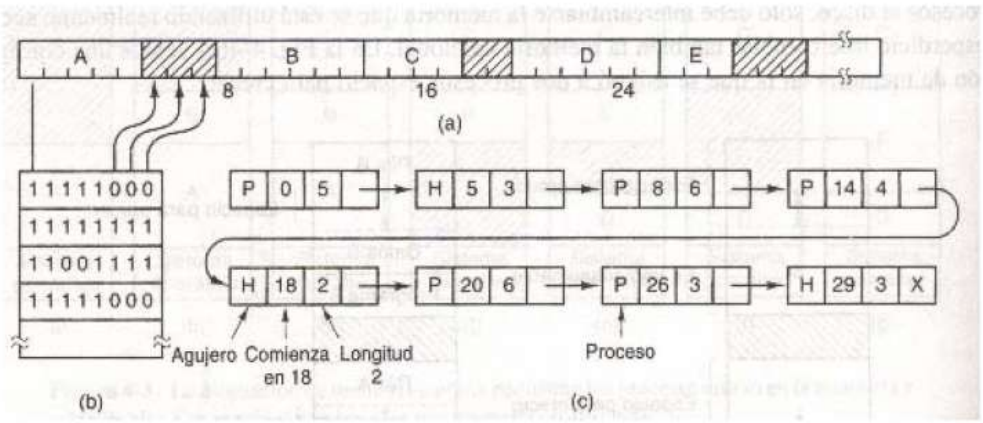


Figura 4-5. (a) Una parte de la memoria con cinco procesos y tres agujeros. Las marcas indican las unidades de asignación de memoria. Las regiones sombreadas (O en el mapa de bits) están libres, (b) El mapa de bits correspondiente, (e) La misma información en forma de lista.

Tabla 6.1 Técnicas de Gestión de Memoria

Técnica	Descripción	Ventajas	Desventajas
Partición fija	La memoria principal se divide en un conjunto de particiones fijas durante la generación del sistema. Un proceso se puede cargar en una partición de mayor o igual tamaño.	Sencilla de implementar; poca sobrecarga del sistema operativo.	Empleo ineficiente de la memoria debido a la fragmentación interna; el número de procesos activos es fijo.
Partición Dinámica	Las particiones se crean dinámicamente, de forma que cada proceso se carga en una partición de exactamente el mismo tamaño que el proceso.	No hay fragmentación interna; uso más eficiente de la memoria principal.	Uso ineficiente del procesador debido a la necesidad de compactación para contrarrestar la fragmentación externa.
Paginación Simple	La memoria principal se divide en un conjunto de marcos de igual tamaño. Cada proceso se divide en una serie de páginas del mismo tamaño que los marcos. Un proceso se carga situando todas sus páginas en marcos libres pero no necesariamente contiguos.	No tiene fragmentación externa.	Hay una pequeña cantidad de fragmentación interna.
Segmentación Simple	Cada proceso se divide en una serie de segmentos. Un proceso se carga situando todos sus segmentos en particiones dinámicas que no tienen por qué ser contiguas.	No tiene fragmentación interna.	Necesita compactación.
Memoria Virtual Paginada	Como la paginación simple, excepto que no hace falta cargar todas las páginas de un proceso. Las páginas no residentes que se necesiten se traerán más tarde, de manera automática.	No hay fragmentación externa; alto grado de multiprogramación; gran espacio virtual para el proceso.	Sobrecarga por gestión compleja de memoria.
Memoria Virtual Segmentada	Como la segmentación simple, excepto que no es necesario cargar todos los segmentos de un proceso. Las segmentos no residentes que se necesiten se traerán más tarde, de forma automática.	No hay fragmentación interna; alto grado de multiprogramación; gran espacio virtual para el proceso; soporte de protección y compartición.	Sobrecarga por gestión compleja de memoria.

18.1.23Espacio de direcciones lógico y físico

Una dirección generada por la CPU se denomina dirección lógica, en tanto que la percibida por la unidad de memoria se conoce como dirección física.

Los esquemas de vinculación de direcciones durante la compilación y durante la carga dan pie a un entorno en el que las direcciones lógicas y físicas son las mismas. En cambio, la ejecución del esquema de vinculación de direcciones durante la ejecución producen un entorno en que las direcciones lógicas y físicas difieren. En este caso, solemos llamar a la dirección lógica dirección virtual. Así pues, en el esquema de vinculación de direcciones en el momento de la ejecución, los espacios de direcciones lógico y físico difieren. La transformación de direcciones virtuales a físicas en el momento de la ejecución corre por cuenta de la unidad de gestión de memoria (MMU). Hay varios esquemas para realizar la transformación(paginacion, segmentacion, paginacion-segmentacion,asignacion con una sola particion), los cuales necesitan soporte de hardware.

18.1.24Intercambio Vs Memoria virtual:

La estrategia más sencilla, llamada **intercambio**, consiste en traer a la memoria cada proceso en su totalidad, ejecutarlo durante un tiempo, y después colocarlo otra vez en el disco. La otra estrategia, llamada **memoria virtual**, permite a los programas ejecutarse aunque sólo estén parcialmente en la memoria principal.

18.1.25Paginacion simple:

[pdf]

La paginación proporciona reubicación dinámica
– la dirección de comienzo de marco juega un papel
equivalente al registro de reubicación
[/pdf]

Tanto las particiones de tamaño fijo como las de tamaño variable hacen un uso ineficiente de la memoria; las primeras generan fragmentación interna, mientras que las segundas originan fragmentación externa. Supóngase, no obstante, que la memoria principal se encuentra particionada en trozos iguales de tamaño fijo relativamente pequeños y que cada proceso está dividido también en pequeños trozos de tamaño fijo y del mismo tamaño que los de memoria. En tal caso, los trozos del proceso, conocidos como páginas, pueden asignarse a los trozos libres de memoria, conocidos como marcos o marcos de página.

Así pues, se puede comprobar que la paginación simple, tal y como se describe, es similar a la partición estática. Las diferencias están en que, con paginación, las particiones son algo más pequeñas, un programa puede ocupar más de una partición y éstas no tienen por qué estar contiguas.

En este caso, la dirección relativa, definida en relación al origen del programa y la dirección lógica, expresada como un número de página y un desplazamiento, son las mismas.

En la figura 6.9 se muestra un ejemplo, donde se emplean direcciones de 16 bits y el tamaño de página es de 1K = 1024 bytes. La dirección relativa 1502 es 0000010111011110 en binario. Con un tamaño de página de 1K, se necesitan 10 bits para el campo de desplazamiento, dejando 6 bits para el número de página. De este modo, un programa puede estar formado por un máximo de 26 = 64 páginas de IKb cada una. Como muestra la figura 6.9b, la dirección relativa 1502 se corresponde con un desplazamiento de 478 (0111011110) en la página 1 (000001), lo que genera el mismo número de 16 bits, 0000010111011110.

Dos son las consecuencias de usar un tamaño de página potencia de dos. Primero, el esquema de direccionamiento lógico es transparente al programador, al ensamblador y al montador. Cada dirección lógica de un programa (número de página, desplazamiento) es idéntica a su dirección relativa. Segundo, resulta relativamente sencillo realizar una función hardware para llevar a cabo la traducción de direcciones dinámicas durante la ejecución. Considérese una dirección de $n + m$ bits, en la que los n bits más significativos son el número de página y los m bits menos significativos son el desplazamiento. En el ejemplo (figura 6.9b), n=6 y m=10. Para la traducción de direcciones hay que dar los siguientes pasos:

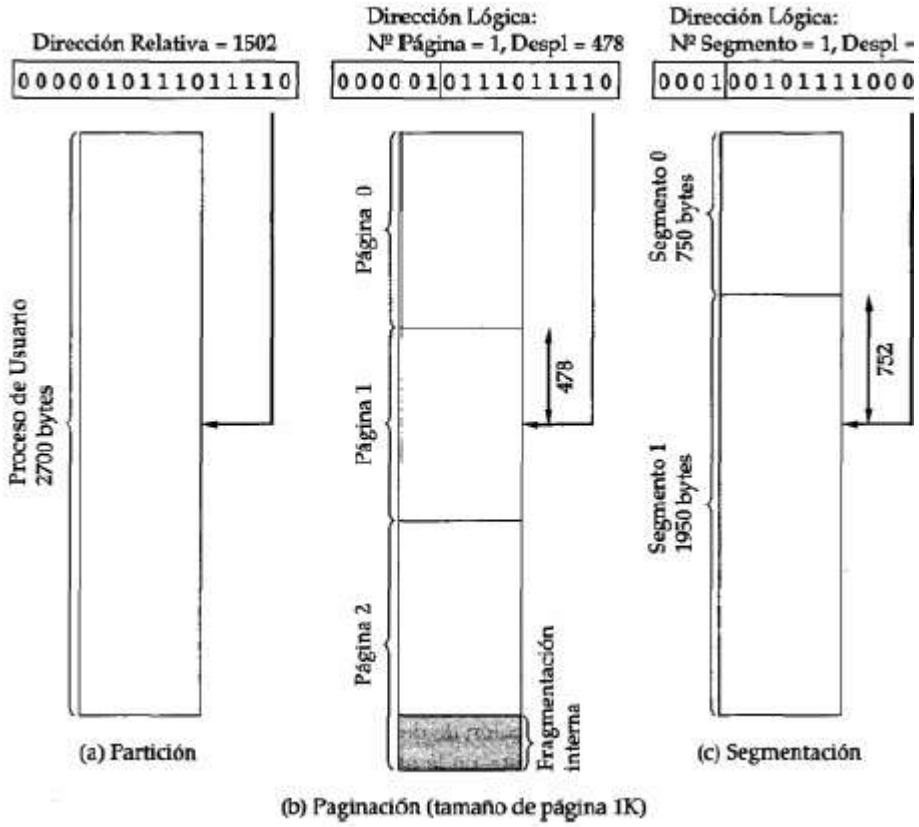
• Obtener el número de página de los n bits más significativos de la dirección lógica.

• Emplear el número de página como índice en la tabla de páginas del proceso para encontrar el número de marco k.

• El comienzo de la dirección física del marco es $k \times 2^m$ y la dirección física del byte referenciado es este número más el desplazamiento. No hace falta calcular esta dirección física, sino que se construye fácilmente concatenando el número de marco con el desplazamiento.

En el ejemplo, se tiene la dirección lógica 0000010111011110, que se corresponde con el número de página 1 y desplazamiento 478. Supóngase que esta página reside en el marco de memoria principal 6 = 000110 en binario. Entonces la dirección física es el marco 6, desplazamiento 478 = 0001100111011110 (figura 6.10a).

Resumiendo, mediante la paginación simple, la memoria principal se divide en pequeños marcos del mismo tamaño. Cada proceso se divide en páginas del tamaño del marco; los pro-cesos pequeños necesitaran pocas páginas, mientras que los procesos grandes necesitarán más. Cuando se introduce un proceso en memoria, se cargan todas sus páginas en los marcos libres y se rellena su tabla de páginas. Esta técnica resuelve la mayoría de los problemas inherentes a la partición.


Dirección Relativa = 1502
0000010111011110
Proceso de Usuario
2700 bytes
(a) Partición
Dirección Lógica:
Nº Página = 1, Despl = 478
000001 0111011110
Página 0
Página 1
Página 2
478
Fragmentación interna
(b) Paginación (tamaño de página 1K)
Dirección Lógica:
Nº Segmento = 1, Despl =
0001 00101111000
Segmento 0
750 bytes
Segmento 1
1950 bytes
752
(c) Segmentación

Cuando un programa ejecuta una instrucción como
MOVE REG,1000

está copiando el contenido de la dirección de memoria 1000 en REG (o viceversa, dependiendo de la
computadora). Las direcciones pueden generarse usando indización, registros de base, registros de segmento y otras técnicas. Estas direcciones generadas por programas se denominan direcciones virtuales y constituyen el espacio de direcciones virtual. En las computadoras sin memoria virtual, la dirección virtual se coloca directamente en el bus de memoria y hace que se lea o escriba la palabra de memoria física que tiene la misma dirección. Cuando se usa memoria virtual, las direcciones virtuales no pasan directamente al bus de memoria; en vez de ello, se envían a una unidad de administración de memoria (MMU), un chip o colección de chips que transforma las direcciones virtuales en direcciones de memoria física

Segmentation is similar to paging but uses variable-sized “pages.”

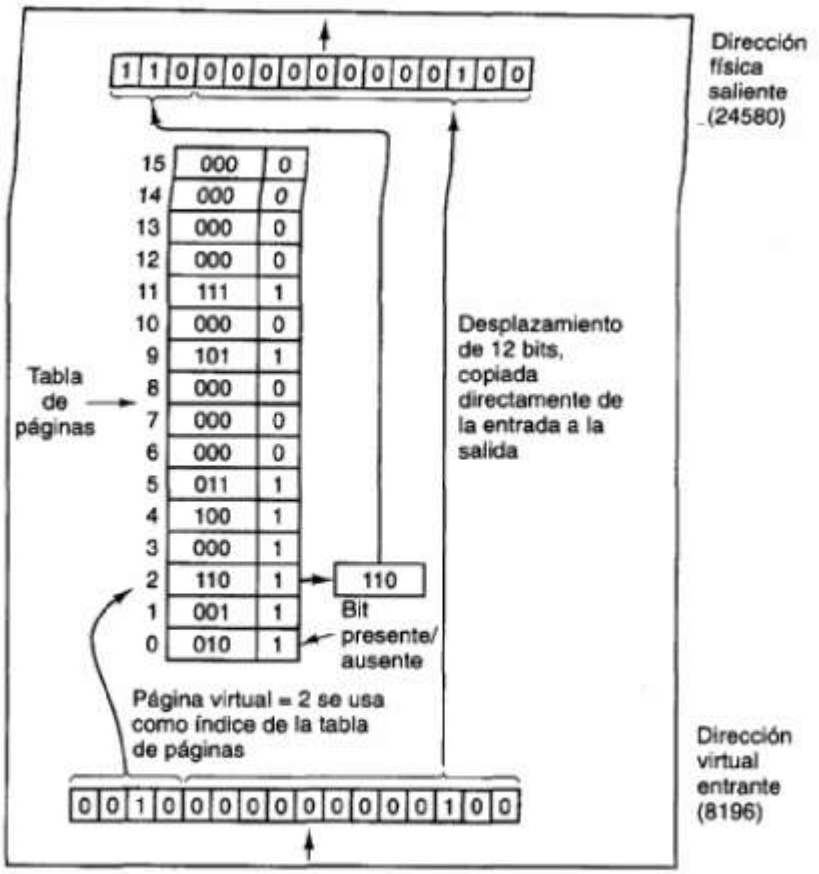


Figura 4-9. Funcionamiento interno de la MMU con 16 páginas de 4K.

18.1.25.1TLB — Buffers de consulta para traducción

En principio, cada referencia a memoria virtual puede generar dos accesos a memoria: uno para obtener la entrada de la tabla de páginas correspondiente y otro para obtener el dato de-seado. Así pues, un esquema sencillo de memoria virtual podría tener el efecto de doblar el tiempo de acceso a memoria. Para solucionar este problema, la mayoría de los esquemas de memoria virtual hacen uso de una cache especial para las entradas de la tabla de páginas, lla-mada generalmente **buffer de traducción adelantada** (TLB, *Translation Lookaside Buffer).* Esta cache funciona del mismo modo que una memoria cache (ver capítulo 1) y contiene aquellas entradas de la tabla de páginas usadas hace menos tiempo. [stal]

En la mayor parte de los esquemas de paginación, las tablas de páginas se mantienen en la memoria,
debido a su gran tamaño. Esto puede tener un impacto enorme sobre el rendimiento. Consideremos, por ejemplo, una instrucción que copia un registro en otro. Si no hay paginación, esta instrucción sólo hace referencia a la memoria una vez, para obtener la instrucción. Si hay paginación, se requieren referencias adicionales a la memoria para acceder a la tabla de páginas. Puesto que la velocidad de ejecución generalmente está limitada por la rapidez con que la CPU puede obtener instrucciones y datos de la memoria, tener que referirse dos veces a la tabla de páginas por cada referencia a la memoria reduce el rendimiento a una tercera parte. En estas condiciones, nadie usaría paginación.

La solución se basa en la observación de que muchos programas tienden a efectuar un gran número de referencias a un número pequeño de páginas, y no al revés. Así, sólo se lee mucho una fracción pequeña de las entradas de la tabla de páginas; el resto apenas si se usa.

La solución que se encontró consiste en equipar las computadoras con un pequeño dispositivo de hardware para transformar las direcciones virtuales en físicas sin pasar por la tabla de páginas. El dispositivo, llamado TLB (buffer de consulta para traducción, en inglés, Translation Lookaside Buffer) o también memoria asociativa, se ilustra en la Fig. 4-12. Generalmente, el TLB está dentro de la MMU y consiste en un pequeño número de entradas, ocho en este ejemplo, pero pocas veces más que 64. Cada entrada contiene información acerca de una página; en particular, el número de página virtual, un bit que se enciende cuando la página se modifica, el código de protección (permisos de leer/escribir/ejecutar) y el marco de página físico en el que se encuentra la página. Estos campos tienen una correspondencia uno a uno con los campos de la tabla de páginas. Otro bit indica si la entrada es válida (si está en uso) o no.

Veamos ahora cómo funciona el TLB. Cuando se presenta una dirección virtual a la MMU para ser traducida, lo primero que hace el hardware es verificar si su número de página virtual está presente en el TLB, comparándolo con todas las entradas simultáneamente (es decir, en paralelo). Si
hay una concordancia válida y el acceso no viola los bits de protección, el marco de página se toma directamente del TLB, sin recurrir a la tabla de páginas. Si el número de página virtual está presente en el TLB pero la instrucción está tratando de escribir en una página sólo de lectura, se generará una falla de protección, tal como sucedería si se usara la tabla de páginas.

El caso interesante es qué sucede si el número de página virtual no está en el TLB. La MMU detecta la falta de concordancia y realiza una búsqueda ordinaria en la tabla de páginas. A continuación desaloja una de las entradas del TLB y la sustituye por la entrada de tabla de páginas que encontró. Por tanto, si esa página se usa pronto otra vez, se le encontrará en el TLB. Cuando una entrada se desaloja del TLB, el bit de modificada se copia en su entrada de tabla de páginas en

la memoria. Los demás valores ya están ahí. Cuando el TLB se carga de la tabla de páginas, se toman todos los campos de la memoria.

Existe una serie de detalles adicionales sobre la organización real de la TLB. Puesto que la TLB contiene sólo algunas de las entradas de la tabla de páginas completa, no se puede indexar simplemente la TLB por el número de página. En su lugar, cada entrada de la TLB debe incluir el número de página, además de la entrada completa a la tabla de páginas. El procesador estará equipado con hardware que permita consultar simultáneamente varias entradas de la TLB para determinar si hay una coincidencia en el número de página. Esta técnica llamada *correspondencia asociativa* y contrasta con la correspondencia directa, que se emplea para buscar en la tabla de páginas de la figura 7.7. El diseñador de la TLB debe tener también en cuenta la forma en que se organizan las entradas en la TLB y qué entrada reemplazar cuando se introduce una nueva. Estas cuestiones deben considerarse en cualquier diseño de cache de hardware. Este problema no se trata aquí; se debe consultar un tratado de diseño de caches para obtener más detalles [sta]

se produce un fallo en el acceso a memoria, llamado **fallo de página.** En este punto, se abandona el ámbito del hardware y se invoca al sistema operativo, que carga la página necesaria y actualiza la tabla de páginas. [sta]

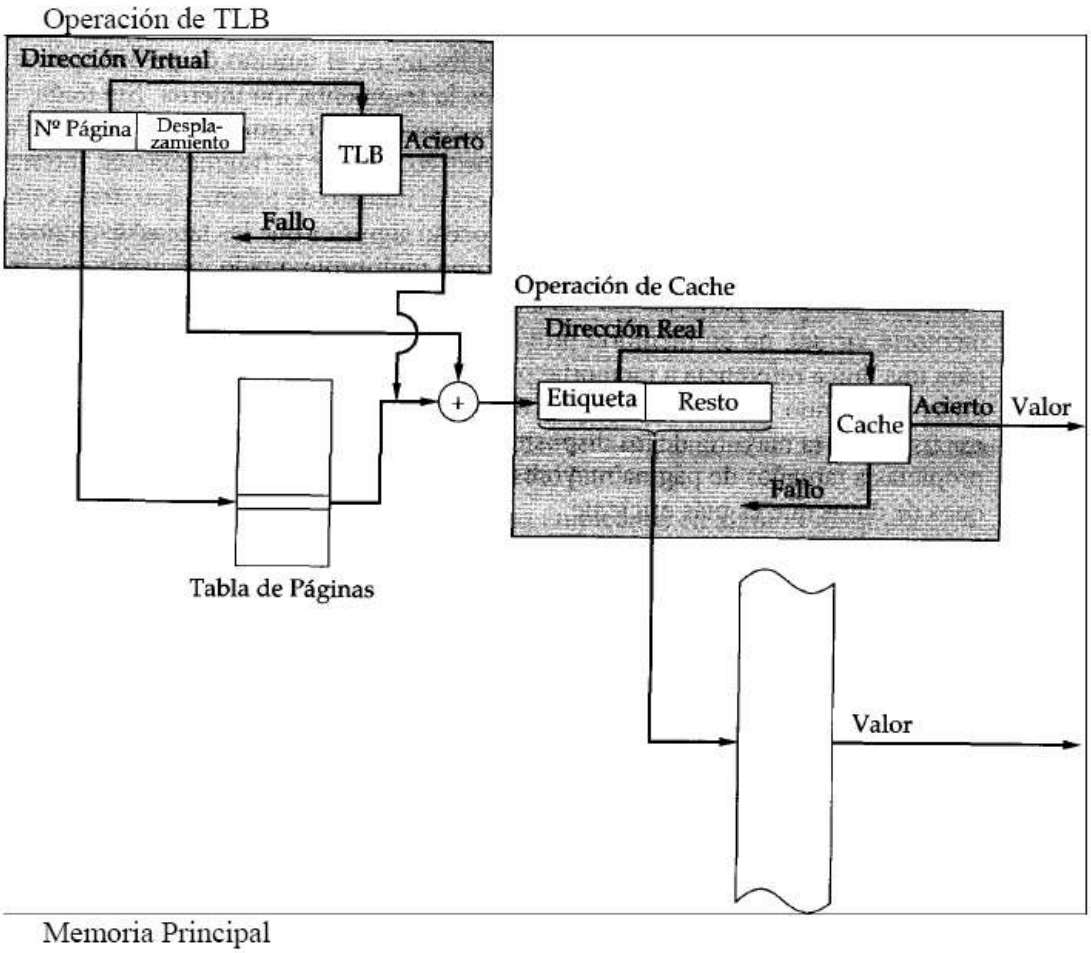


FIGURA 7.8 Funcionamiento del buffer de traducción adelantada y la cache

Administración de TLB por software

Hasta aquí, hemos supuesto que todas las máquinas con memoria virtual paginada tienen tablas de páginas
reconocidas por el hardware, más un TLB. En este diseño, la administración del TLB y el manejo de fallas
de TLB corren por cuenta exclusivamente del hardware de la MMU. Las trampas al sistema operativo sólo
ocurren cuando una página no está en la memoria.
En el pasado, este supuesto era cierto, pero algunas máquinas RISC modernas, incluidas MIPS, Alpha y HP PA, realizan casi toda esta administración de páginas en software. En estas máquinas, el
sistema operativo carga explícitamente las entradas del TLB. Cuando no se encuentra una página virtual

en el TLB, en lugar de que la MMU se remita simplemente a las tablas de páginas para encontrar y obtener la referencia de página requerida, genera una falla de TLB y deja que el sistema operativo se
encargue del problema. Éste debe encontrar la página, eliminar una entrada del TLB, introducir la nueva y
reiniciar la instrucción que falló. Y, desde luego, todo esto debe hacerse con unas cuantas instrucciones,
porque las fallas de TLB podrían ocurrir con mucha mayor frecuencia que las fallas de página.

18.1.26Segmentación:

Otro modo de subdividir el programa es la segmentación. En este caso, el programa y sus datos asociados se dividen en un conjunto de **segmentos.** No es necesario que todos los seg-mentos de todos los programas tengan la misma longitud, aunque existe una longitud máxima de segmento. Como en la paginación, una dirección lógica segmentada consta de dos partes, en este caso un número de segmento y un desplazamiento.

Como consecuencia del empleo de segmentos de distinto tamaño, la segmentación resulta similar a la partición dinámica. En ausencia de un esquema de superposición o del uso de memoria virtual, sería necesario cargar en memoria todos los segmentos de un programa para su ejecución. La diferencia, en comparación con la partición dinámica, radica en que, con segmentación, un programa puede ocupar más de una partición y éstas no tienen por qué estar contiguas. La segmentación elimina la fragmentación interna, pero, como la partición dinámica, sufre de fragmentación externa. Sin embargo, debido a que los procesos se dividen en un conjunto de partes más pequeñas, la fragmentación externa será menor.

Mientras que la paginación es transparente al programador, la segmentación es generalmente visible y se proporciona como una comodidad para la organización de los programas y datos. Normalmente, el programador o el compilador asigna los programas y los datos a diferentes segmentos. En aras de la programación modular, el programa o los datos pueden ser divididos de nuevo en diferentes segmentos. El principal inconveniente de este servicio es que el programador debe ser consciente de la limitación de tamaño máximo de los segmentos.

Otra consecuencia del tamaño desigual de los segmentos es que no hay una correspondencia simple entre las direcciones lógicas y las direcciones físicas. De forma análoga a la paginación, un esquema de segmentación simple hará uso de una tabla de segmentos para cada proceso y una lista de bloques libres en memoria principal. Cada entrada de tabla de segmentos tendría que contener la dirección de comienzo del segmento correspondiente en memoria principal. La entrada deberá proporcionar también la longitud del segmento para asegurar que no se usan direcciones no válidas. Cuando un proceso pasa al estado Ejecutando, se carga la dirección de su tabla de segmentos en un registro especial del hardware de gestión de memoria. Considérese una dirección de $n + m$ bits, en la que los n bits más significativos son el número de segmento y los m bits menos significativos son el desplazamiento. En el ejemplo (figura 6.9c), $n=4$ y $m= 12$. Así pues, el máximo tamaño de segmento es $2^{12} = 4096$. Para la traducción de direcciones hay que dar los siguientes pasos:

- Extraer el número de segmento de los n bits más significativos de la dirección lógica.
- Emplear el número de segmento como índice en la tabla de segmentos del proceso para encontrar la dirección física de comienzo del segmento.
- Comparar el desplazamiento, expresado por los m bits menos significativos, con la longitud del segmento. Si el desplazamiento es mayor que la longitud, la dirección no es válida.

• La dirección física buscada es la suma de la dirección física de comienzo del segmento más el desplazamiento.

En el ejemplo, se emplea la dirección lógica 0001001011110000, que se corresponde con el número de segmento 1 y desplazamiento 752. Supóngase que dicho segmento está en memoria principal y comienza en la dirección física 0010000000100000. Entonces la dirección física será 0010000000100000 + 001011110000 = 0010001100010000.

En definitiva, con segmentación simple, un proceso se divide en varios segmentos que no tienen por qué ser del mismo tamaño. Cuando un proceso se introduce en memoria, se cargan todos sus segmentos en regiones de memoria libres y se rellena la tabla de segmentos.

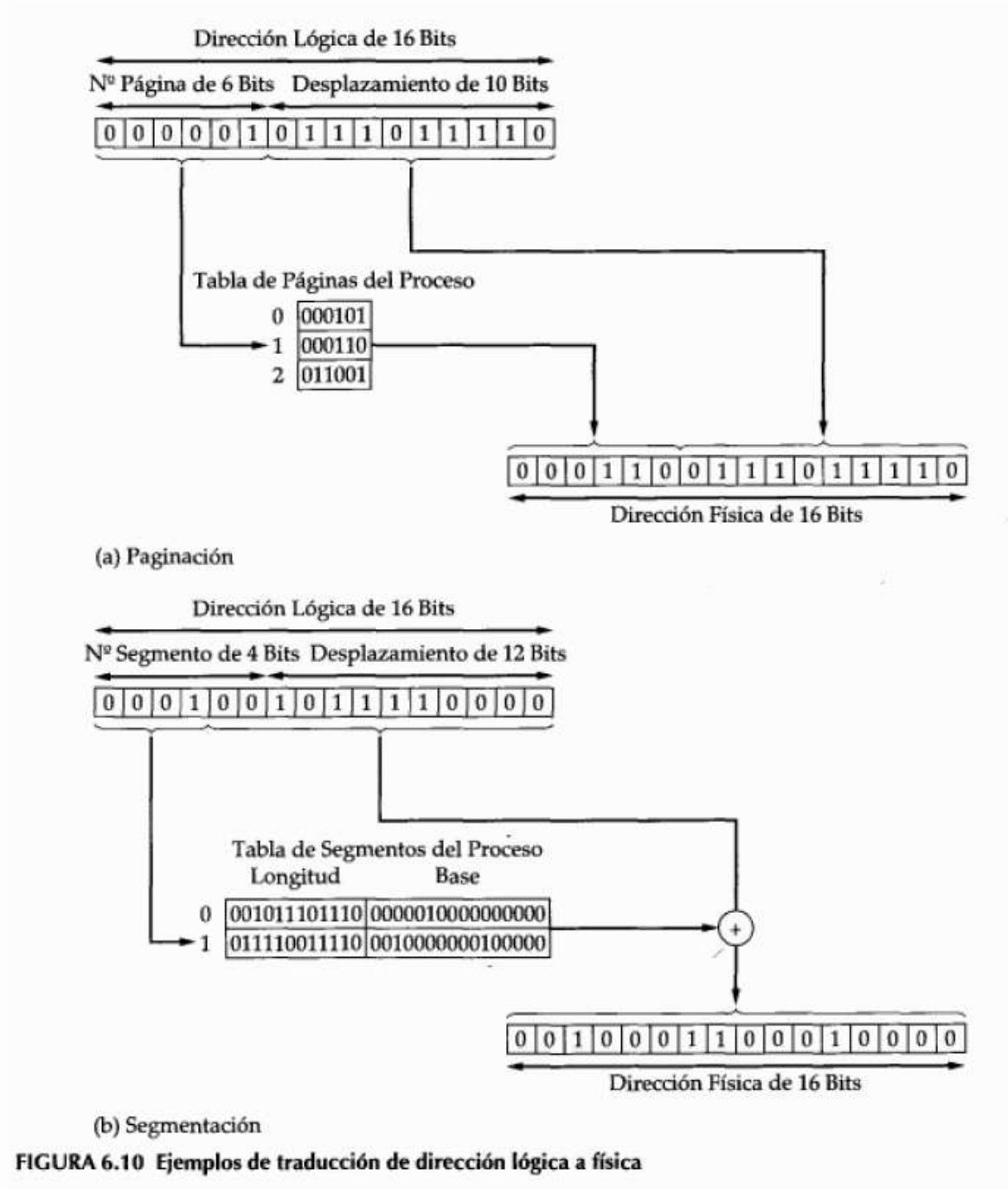


FIGURA 6.10 Ejemplos de traducción de dirección lógica a física

19 Memoria Virtual:

ESTRUCTURAS DE HARDWARE Y DE CONTROL

Cuando se compararon la paginación y segmentación simple por un lado, con la partición estática y dinámica por otro, se pusieron las bases de un avance fundamental en la gestión de

memoria. Las claves de este avance son dos características de la paginación y la segmentación:
1. Todas las referencias a memoria dentro de un proceso son direcciones lógicas que se tra-ducen dinámicamente a direcciones físicas durante la ejecución. Esto quiere decir que un proceso puede cargarse y descargarse de memoria principal de forma que ocupe regiones diferentes en instantes diferentes a lo largo de su ejecución.
2. Un proceso puede dividirse en varias partes (páginas o segmentos) y no es necesario que estas partes se encuentren contiguas en memoria principal durante la ejecución. Esto es posible por la combinación de la traducción dinámica de direcciones en tiempo de ejecución y el uso de una tabla de páginas o de segmentos.

Volviendo sobre el avance comentado, si *estas dos características están presentes, no será necesario que todas las páginas o todos los segmentos de un proceso estén en memoria durante la ejecución.* Si tanto la parte (página o segmento) que contiene la siguiente instrucción a leer como la parte que contiene los próximos datos a acceder están en memoria principal, la ejecución podrá continuar al menos por un tiempo.

Se llamará **conjunto residente** del proceso a la parte de un proceso que está realmente en memoria principal.

Cuando el proceso se ejecuta,
todo irá sobre ruedas mientras todas las referencias a memoria estén en posiciones que pertenezcan al conjunto residente. A través de la tabla de páginas o de segmentos, el procesador siempre es capaz de determinar si esto es así. Si el procesador encuentra una dirección lógica que no está en memoria principal, genera una interrupción que indica un fallo de acceso a memoria. El sistema operativo pasa el proceso interrumpido al estado bloqueado y toma el control. Para que la ejecución de este proceso siga más tarde, el sistema operativo necesita traer a memoria principal el fragmento del proceso que contiene la dirección lógica que provocó el fallo de acceso. Para ello, el sistema operativo emite una solicitud de Lectura de E/S a disco. Después de haber emitido la solicitud de E/S, el sistema operativo puede despachar otro proceso para que se ejecute mientras se realiza la operación de E/S. Una vez que el fragmento deseado se ha traído a memoria principal y se ha emitido la interrupción de E/S, se devuelve el control al sistema operativo, que coloca el proceso afectado en el estado de Listo.

Ventajas:

1. Se pueden conservar más procesos en memoria principal. Puesto que se van a cargar sólo algunos fragmentos de un proceso particular, habrá sitio para más procesos. Esto conduce a una utilización más eficiente del procesador, puesto que es más probable que, por lo menos, uno de los numerosos procesos esté en estado Listo en un instante determinado.

2. Es posible que un proceso sea más grande que toda la memoria principal. Se elimina así una de las limitaciones más notorias de la programación. Sin el esquema que se ha expuesto, un programador debe ser consciente de cuánta memoria tiene disponible. Si el programa que está escribiendo es demasiado grande, el programador debe idear formas de estructurar el programa en fragmentos que puedan cargarse de forma separada con algún tipo de estrategia de superposición. Con una memoria virtual basada en paginación o segmentación, este trabajo queda para el sistema operativo y el hardware. En lo que atañe al programador, se las arregla con una memoria enorme, dependiendo del tamaño de almacenamiento en disco. El sistema operativo cargará automáticamente en memoria principal los fragmentos de un proceso cuando los necesita.

-De este modo, en un instante dado, en memoria sólo se tienen unos pocos fragmentos de un proceso dado y, por tanto, se pueden mantener más procesos en memoria. Es más, se ahorra tiempo, porque los fragmentos que no se usan no se cargan ni se descargan de memoria.

-Así pues, cuando el sistema operativo traiga a memoria un fragmento, deberá expulsar otro. Si expulsa un fragmento justo antes de ser usado, tendrá que traer de nuevo el fragmento de manera casi inmediata. Demasiados intercambios de fragmentos conducen a lo que se conoce como **hiperpaginación** *(thrashing):* El procesador consume más tiempo intercambiando fragmentos que ejecutando instrucciones de usuario.

-Soluciones a la hiperpaginacón:

En esencia, el sistema operativo intenta adivinar, en función de la historia reciente, qué fragmentos se usarán con menor probabilidad en un futuro próximo. Los argumentos anteriores se basan en el principio de cercanía; el principio de cercanía afirma que las referencias a los datos y al programa dentro de un proceso tienden a agruparse. Por tanto, es válida la suposición de que, durante cortos periodos, se necesitarán sólo unos pocos fragmentos de un proceso.

TABLA 7.1 Características de la Paginación y la Segmentación

Paginación Simple	Segmentación Simple	Memoria Virtual Paginada	Memoria Virtual Segmentada
La memoria principal está dividida en trozos pequeños de tamaño fijo llamados marcos.	La memoria principal no está dividida.	La memoria principal está dividida en trozos pequeños de tamaño fijo llamados marcos.	La memoria principal no está dividida.
El compilador o el sistema gestor de memoria dividen los programas en páginas.	El programador especifica al compilador los segmentos del programa (es decir, el programador toma la decisión).	El compilador o el sistema gestor de memoria dividen los programas en páginas.	El programador especifica al compilador los segmentos del programa (es decir, el programador toma la decisión).
Fragmentación interna en los marcos.	No hay fragmentación interna.	Fragmentación interna en los marcos.	No hay fragmentación interna.
No hay fragmentación externa.	Fragmentación externa.	No hay fragmentación externa.	Fragmentación externa.
El sistema operativo debe mantener una tabla de páginas para cada proceso, indicando en qué marco está cada página.	El sistema operativo debe mantener una tabla de segmentos para cada proceso, indicando la dirección de carga y la longitud de cada segmento.	El sistema operativo debe mantener una tabla de páginas para cada proceso, indicando en qué marco está cada página.	El sistema operativo debe mantener una tabla de segmentos para cada proceso, indicando la dirección de carga y la longitud de cada segmento.
El sistema operativo debe mantener una lista de marcos libres.	El sistema operativo debe mantener una lista de huecos libres en memoria principal.	El sistema operativo debe mantener una lista de marcos libres.	El sistema operativo debe mantener una lista de huecos libres en memoria principal.
El procesador emplea número de página y desplazamiento para calcular las direcciones absolutas.	El procesador emplea número de segmento y desplazamiento para calcular las direcciones absolutas.	El procesador emplea número de página y desplazamiento para calcular las direcciones absolutas.	El procesador emplea número de segmento y desplazamiento para calcular las direcciones absolutas.
Todas las páginas de un proceso deben estar en memoria principal para que el proceso ejecute, a menos que se use superposición.	Todos los segmentos de un proceso deben estar en memoria principal para que el proceso ejecute, a menos que se use superposición.	Para que el proceso ejecute, no hace falta que todas sus páginas estén en marcos de memoria principal. Las páginas se leerán cuando se necesiten. La carga de una página en memoria principal puede exigir descargar otra al disco.	Para que el proceso ejecute, no hace falta que todos sus segmentos estén en memoria principal. Los segmentos se leerán cuando se necesiten. La carga de un segmento en memoria principal puede requerir descargar uno o más segmentos al disco.

Así pues, se puede comprobar que el principio de cercanía sugiere que los esquemas de memoria virtual pueden funcionar. Para que la memoria virtual sea práctica y efectiva, se necesitan dos

ingredientes. Primero, debe haber un soporte de hardware y, en segundo lugar, el sistema operativo debe incluir un software para gestionar el movimiento de páginas y/o segmentos entre memoria secundaria y memoria principal.

19.1.1 Paginación

En el estudio de la paginación simple se indicó que cada proceso tiene su propia tabla de páginas y que, cuando carga todas sus páginas en memoria principal, se crea y carga en memoria principal una tabla de páginas. Cada entrada de la tabla de páginas contiene el número de marco de la página correspondiente en memoria principal. Cuando se considera un esquema de memoria virtual basado en la paginación se necesita la misma estructura, una tabla de páginas.

Puesto que sólo algunas de las páginas de un proceso pueden estar en memoria principal, se necesita un bit en cada entrada de la tabla para indicar si la página correspondiente está presente (P) en memoria principal o no lo está. Si el bit indica que la página está en memoria, la entrada incluye también el número de marco para esa página.

Otro bit de control necesario en la entrada de la tabla de páginas es el bit de modificación M), para indicar si el contenido de la página correspondiente se ha alterado desde que la página se cargó en memoria principal. Si no ha habido cambios, no es necesario escribir la página cuando sea sustituida en el marco que ocupa actualmente. Puede haber también otros bits de control.Por ejemplo, si la protección o la compartición se gestiona a nivel de página, se necesitarán más bits con tal propósito.

19.1.2 Estructura de la Tabla de Páginas

[tan]
Pasemos ahora de la estructura macroscópica de las tablas de páginas a los detalles de una entrada de tabla de páginas. La organización exacta de una entrada depende en gran medida de la máquina, pero la clase de información presente es aproximadamente la misma en todas las máquinas. En la Fig. 4-11 presentamos un ejemplo de entrada de tabla de páginas. El tamaño varia de una computadora a otra, pero un tamaño común es de 32 bits. El campo más importante es el Número de marco de página. Después de todo, el objetivo de la transformación de páginas es localizar este valor. Junto a él tenemos el bit Presente/ausente.
Si este bit es 1, la entrada es válida y puede usarse; si es O, la página virtual a la que la entrada pertenece no está actualmente en la memoria. El acceso a una entrada de tabla de páginas que tiene este bit en O causa una falla de página.
Los bits de Protección indican qué clases de acceso están permitidas. En la forma más sencillla, este campo contiene un bit, y O significa lectura/escritura mientras que 1 significa sólo lectura.

Una organización más compleja tendría tres bits, para habilitar la lectura, escritura y ejecución, respectivamente.
Los bits Modificada y Referida siguen la pista al empleo de la página. Cuando se escribe en una

página, el hardware automáticamente enciende el bit Modificada. Este bit es útil cuando el sistema operativo decide reutilizar un marco de página. Si la página que contiene ha sido modificada (es decir, está "sucia"), se deberá escribir de vuelta en el disco. Si la página no ha sido modificada (si está "limpia"), puede abandonarse, ya que la copia que está en el disco aún es válida. Este bit también se conoce como bit sucio, pues refleja el estado de la página.
El bit Referida se enciende cada vez que se hace referencia a una página, sea para lectura o escritura. Su propósito es ayudar al sistema operativo a escoger la página que desalojará cuando ocurra una falla de página. Las páginas que no se están usando son mejores candidatos que las que están en uso, y este bit desempeña un papel importante en varios de los algoritmos de reemplazo de páginas que estudiaremos más adelante en este capítulo.
El último bit permite inhabilitar la colocación en caché de la página. Esta capacidad es importante en el caso de páginas que corresponden a registros de dispositivos en lugar de memoria. Si el sistema operativo se queda dando vueltas en un ciclo esperando que un dispositivo de E/S responda a un comando que se le acaba de dar, es indispensable que el hardware continúe obteniendo la palabra del dispositivo y no use una copia vieja guardada en caché. Con este bit, puede desactivarse la colocación en caché. Las máquinas que tienen un espacio de E/S independiente y no usan E/S con mapa en la memoria no necesitan este bit.
Adviértase que la dirección en disco en la que está guardada la página cuando no está en memoria no forma parte de la tabla de páginas. La razón es sencilla. La tabla de páginas contiene sólo la información que el hardware necesita para traducir una dirección virtual en una dirección física. La información que el sistema operativo necesita para manejar las fallas de página se mantiene en tablas de software dentro del sistema operativo.

[sta]

-Puesto que la tabla de páginas es de longitud variable, en función del tamaño del proceso, no es posible suponer que quepa en los registros. En su lugar, debe estar en memoria principal para ser accesible. La figura 7.3 sugiere una implementación en hardware de este esquema.

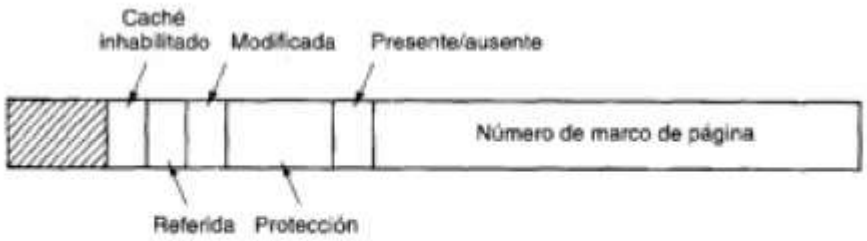


Figura 4-11. Entrada de tabla de páginas representativa.

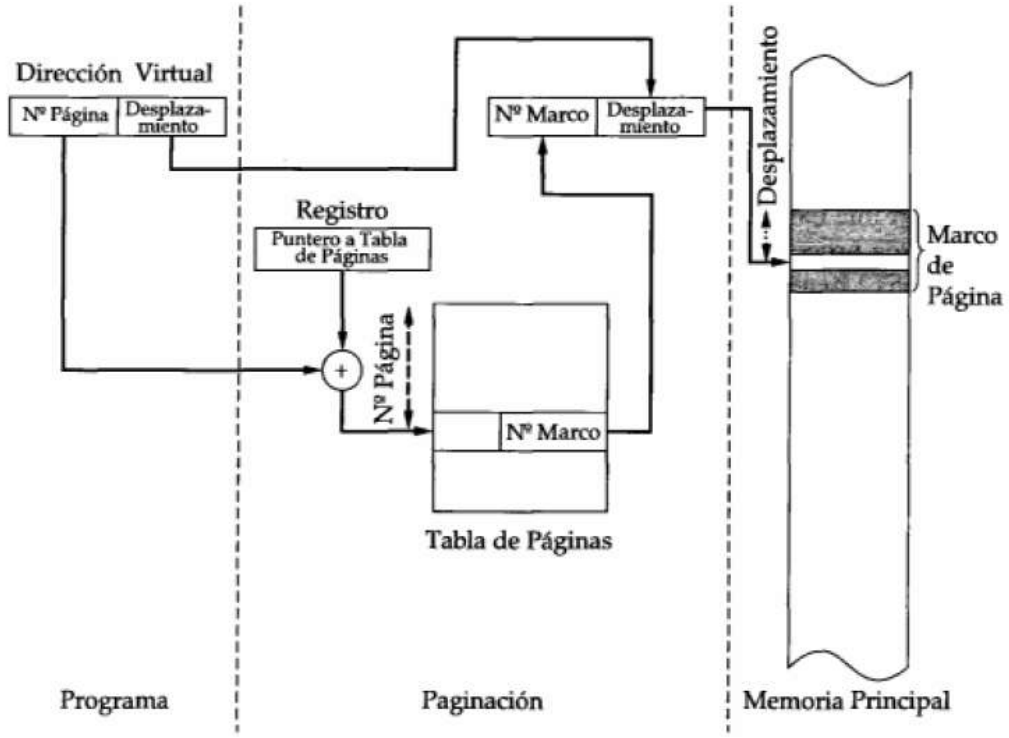
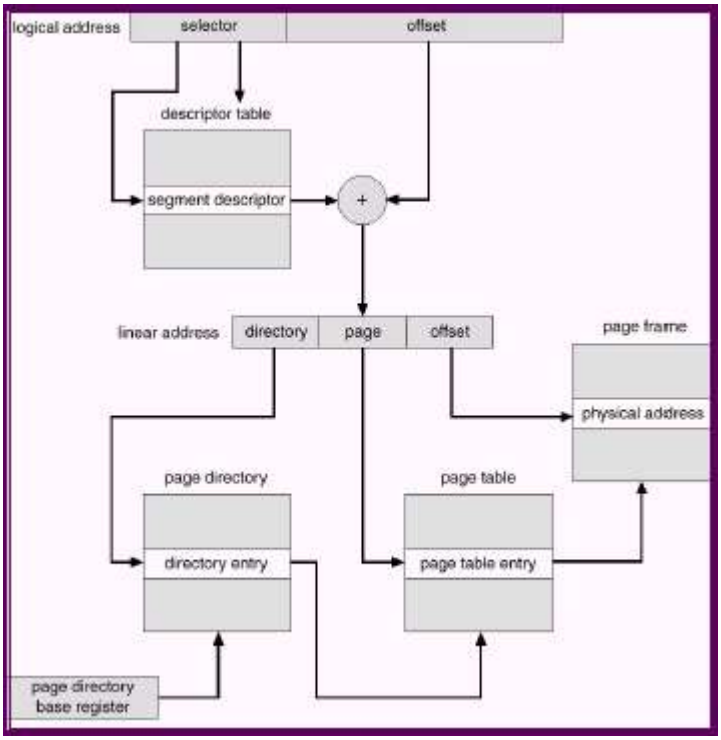


FIGURA 7.3 Traducción de direcciones en un sistema de paginación

Cuando se esta ejecutando un proceso en particular, la dirección de comienzo de la tabla de páginas para este proceso se mantiene en un registro. El número de página de la dirección virtual se emplea como índice en esta tabla para buscar el número de marco correspondiente. Este se combina con la parte de desplazamiento de la dirección virtual para generar la dirección real deseada.

la cantidad de memoria dedicada sólo a la tabla de páginas puede ser inaceptable. Para solucionar este problema, la mayoría de los esquemas de memoria virtual almacenan las tablas de páginas en memoria virtual en vez de en memoria real. Esto significa que estas tablas de páginas están también sujetas a paginación, de la misma forma que las otras páginas.

Algunos procesadores usan un esquema a dos niveles para organizar grandes tablas de páginas. En este esquema, hay un directorio de páginas en el que cada entrada señala a una tabla de páginas. Así pues, si la longitud del directorio de páginas es *X* y la longitud máxima de una tabla de páginas es *Y,* un proceso puede estar formado por hasta *X* x *Y* páginas.



9.17 Consider the Intel address translation scheme shown in Figure 9.20.
a. Describe all the steps that the Intel 80386 takes in translating a logical address into a physical address.
b. What are the advantages to the operating system of hardware that provides such complicated memory translation hardware?
c. Are there any disadvantages to this address translation system?
Answer:
a. The selector is an index into the segment descriptor table. The segment descriptor result plus the original offset is used to produce a linear address with a dir, page, and offset. The dir is an index into a page directory. The entry from the page directory selects the page table, and the page field is an index into the page table. The entry from the page table, plus the offset, is the physical address.
b. Such a page translation mechanism offers the flexibility to allow most operating systems to implement their memory scheme in hardware, instead of having to implement some parts in hardware and some in software. Because it can be done in hardware, it is more efficient (and the kernel is simpler).
c. Address translation can take longer due to the multiple table lookups it can invoke. Caches help, but there will still be cache misses.

Un enfoque alternativo al uso de tablas de páginas a uno o dos niveles consiste en el uso de una estructura de **tabla de páginas invertida .**

Con este método, la parte del número de página en una dirección virtual se contrasta en una tabla de dispersión por medio de una función de dispersión simple'. La tabla de dispersión contiene un puntero a la tabla de páginas invertida, que contiene a su vez las entradas de la tabla de páginas. Con esta estructura, hay una entrada en la tabla de dispersión y la tabla de páginas invertida por cada página de memoria real en lugar de una por cada página virtual. Así pues, necesita una parte fija de la memoria real para las tablas, sin reparar en el número de procesos de páginas virtuales

soportados. Puesto que más de una dirección de memoria pueden corresponderse con la misma entrada de la tabla de dispersión, para gestionar las colisiones emplea una técnica de encadenamiento. La técnica de dispersión genera normalmente cadenas cortas, de dos a tres entradas cada una.[sta]

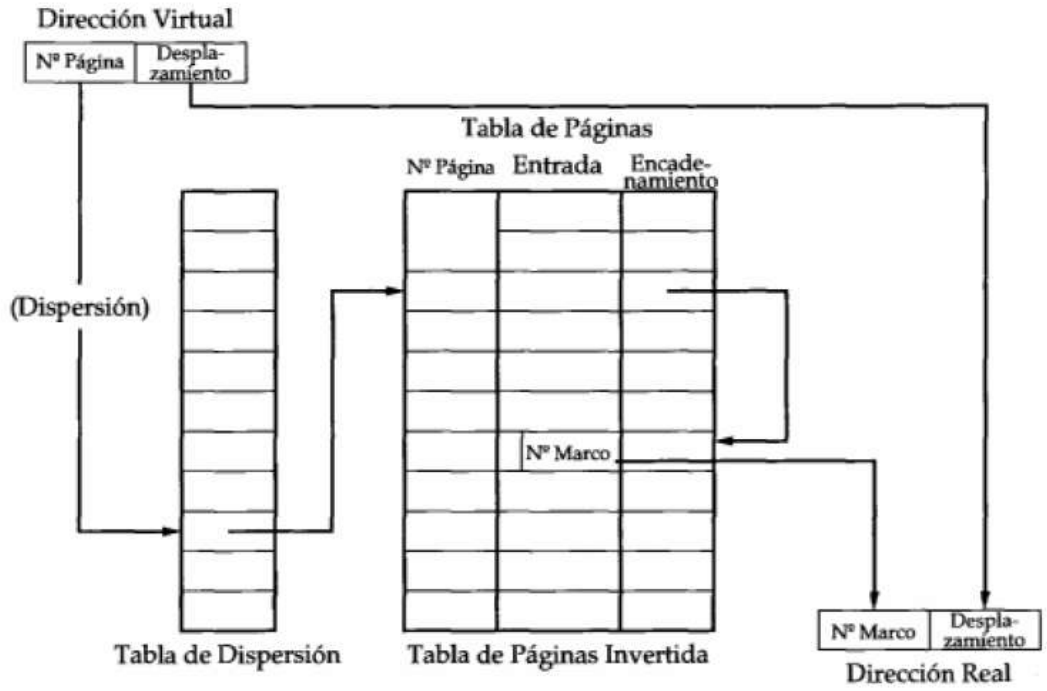


FIGURA 7.4 Estructura de la tabla de páginas invertida

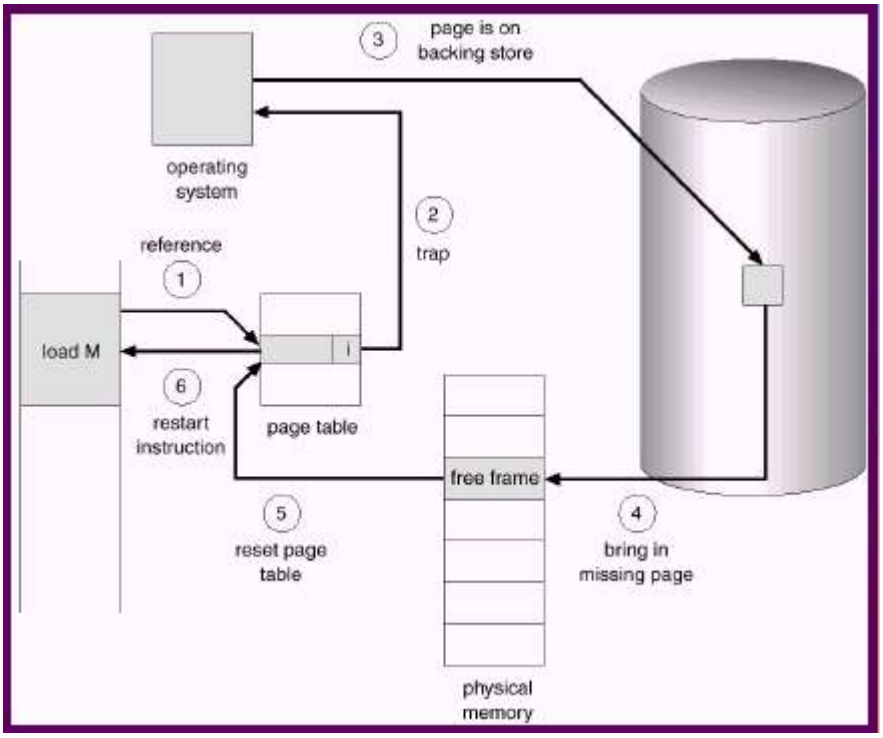
Por lo regular cada proceso tiene su propia tabla de paginas, la cual tiene una entrada por cada página que el proceso está usando (o una ranura por cada dirección virtual, sin importar la validez de esta última). Una de las desventajas de este esquema es que cada tabla de paginas podría contener millones de entradas, Las tablas podrían consumir grandes cantidades de memoria física, que se ocupa unicamente de mantenerse al tanto del resto de la memoria fisica. Para resolver este problema, podemos usar una tabla de páginas invertida. Una tabla de páginas invertida tiene una entrada por cada página (marco) real de memoria. Cada entrada consiste en la dirección virtual de la página almacenada en esa posición de memoria real, con información acerca del proceso dueño de esa página. Así, sólo hau una tala de paginas en el sistema, y sólo tiene una entrada por cada página de memoria física.
Aunque este esquema reduce la cantidad de memoria necesaria para almacenar las tablas de paginas, aumenta el tiempo que toma realizar la búsqueda en la tabla cuando ocurre una referencia a una página. Puesto que la tabla está ordenada por dirección física, y lo que se busca es una dirección virtual, cabe la posibilidad de tener que recorrer toda la tabla para encontrar un valor coincidente; Por esta razon se utiliza una tabla de dispersión (hashing).[silb]

To reduce the size of this data structure, some computers apply a hashing function to the virtual address. The hash allows the data structure to be the length of the number of physical pages in main memory. This number could be much smaller than the number of virtual pages. Such a structure is called an inverted page table. [patt]

Las tablas de páginas tradicionales del tipo que hemos descrito hasta ahora requieren una entradas por cada página virtual, ya que están indizadas por número de página virtual. Si el espacio de direcciones consta de 232 bytes, con 4096 bytes por página, se necesitará más de un millón de entradas de tabla.
Como mínimo, la tabla de páginas tendrá que ocupar 4 megabytes. En los sistemas grandes, este tamaño probablemente será manejable. Sin embargo, al hacerse cada vez más comunes las computadoras de 64 bits, la situación cambia drásticamente. Si el espacio de direcciones ahora tiene 264 bytes, con páginas de 4K, necesitaremos más de 1015 bytes para la tabla de páginas. Sacar de circulación un millón de gigabytes sólo para la tabla de páginas no es razonable, ni ahora, ni durante varias décadas más, sí es que alguna vez lo es. Por tanto, se requiere una solución distinta para los espacios de direcciones virtuales de 64 bits paginados.
Una solución es la tabla de páginas invertida. En este diseño, hay una entrada por marco de página de la memoria real, no por cada página del espacio de direcciones virtual. Por ejemplo, con drecciones virtuales de 64 bits, páginas de 4K y 32MB de RAM, una tabla de páginas invertida sólo requiere 8192 entradas. La entrada indica cuál (proceso, página virtual) está en ese marco de página.
Aunque las tablas de páginas invertidas ahorran enormes cantidades de espacio, al menos cuando el espacio de direcciones virtual es mucho más grande que la memoria física, tienen una desventaja importante: la traducción de virtual a física se vuelve mucho más difícil. Cuando el proceso n hace referencia a la página virtual p, el hardware ya no puede encontrar la página física usando p como índice de la tabla de páginas. En vez de ello, debe buscar en toda la tabla de páginas invertida una entrada (n, p).
Además, esta búsqueda debe efectuarse en cada referencia a la memoria, no sólo cuando hay una falla de página. Examinar una tabla de 8K cada vez que se hace referencia a la memoria no es la forma de hacer que una máquina sea vertiginosamente rápida.
La solución a este dilema es usar el TLB. Si el TLB puede contener todas las páginas de uso

pesado, la traducción puede efectuarse con tanta rapidez como con las tablas de páginas normales. Sin embargo, si ocurre una falla de TLB, habrá que examinar la tabla de páginas invertida. Si se usa una tabla de dispersión como índice de la tabla de páginas invertida, esta búsqueda puede hacerse razonablemente rápida. Ya se están usando tablas de páginas invertidas en algunas estaciones de trabajo IBM y Hewlett-Packard, y se harán más comunes conforme se generalice el uso de máquinas de 64 bits. [tan]

19.1.3 Pasos a seguir para atender un fallo de pagina :



En el contexto de un sistema de paginación por demanda, sucede cuando un proceso intenta referenciar una página que no se incorporó a memoria. El hardware de paginación, al traducor la dirección , observa que la entrada correspondiente de la tabla tiene indicada esa página como inválida, y genera una trap para el sistema operativo (la instrucción que hizo la referencia no se completa).

Entonces el sistema

1. determina si la referencia memoria fue válida o inválida (por ejemplo por indizar mal un vector)
2. Si fue una referencia válida, en vez de abortar el proceso, localizar en disco la pagina buscada
3. Buscar un marco libre
    a. si lo hay, lo selecciona

   b. sino, aplica un algoritmo de reemplazo para seleccionar una página a quitar, la guarda en disco y refleja el cambio en la tabla de páginas.
4. Mover la página deseada al nuevo marco libre, reflejar el cambio en la tabla de paginas
5. Reejecutar la intrucción que provocó el fallo (el proceso no necesariamente cambio de estado).

Obs. Se planifica (CPU) en las transferencias entre disco y memoria de la página seleccionada para reemplazo y de la página entrante de acuerdo a las reglas habituales para E/S.

Proteccion y compartición

Why is it that, on a system with paging, a process cannot access memory it does not own? How could the operating system allow access to other memory? Why should it or should it not?
Answer: An address on a paging system is a logical page number and an offset. The physical page is found by searching a table based on the logical page number to produce a physical page number. Because the operating system controls the contents of this table, it can limit a process to accessing only those physical pages allocated to the process. There is no way for a process to refer to a page it does not own because the page will not be in the page table. To allow such access, an operating system simply needs to allow entries for non-process memory to be added to the process's page table. This is useful when two or more processes need to exchange data—they just read and write to the same physical addresses (which may be at varying logical addresses). This makes for very efficient interprocess communication.

What is the effect of allowing two entries in a page table to point to the same page frame in memory? Explain how this effect could be used to decrease the amount of time needed to copy a large amount of memory from one place to another. What effect would updating some byte on the one page have on the other page?
Answer: By allowing two entries in a page table to point to the same page frame in memory, users can share code and data. If the code is reentrant, much memory space can be saved through the shared use of large programs such as text editors, compilers,and database systems. "Copying" large amounts of memory could be effected by having different page tables point to the same memory location.
However, sharing of nonreentrant code or data means that any user having access to the code can modify it and these modifications would be reflected in the other user's "copy."

Describe a mechanism by which one segment could belong to the address space of two different processes.
Answer: Since segment tables are a collection of base–limit registers, segments can be shared when entries in the segment table of two different jobs point to the same physical location. The two segment tables must have identical base pointers, and the shared segment number must be the same in the two processes.

Explain why it is easier to share a reentrant module using segmentation than it is to do so when pure paging is used.
Answer: Since segmentation is based on a logical division of memory rather than a physical one, segments of any size can be shared with only one entry in the segment tables of each user. With paging there must be a common entry in the page tables for each page that is shared.

Sharing segments among processes without requiring the same segment number is possible in a dynamically linked segmentation system.

a. Define a system that allows static linking and sharing of segments without requiring that the segment numbers be the same.
b. Describe a paging scheme that allows pages to be shared without requiring that the page numbers be the same.
Answer: Both of these problems reduce to a program being able to reference both its own code and its data without knowing the segment or page number associated with the address. MULTICS solved this problem by associating four registers with each process. One register had the address of the current program segment, another had a base address for the stack, another had a base address for the global data, and so on. The idea is that all references have to be indirect through a register that maps to the current segment or page number. By changing these registers, the same code can execute for different processes without the same page or segment numbers.

Discuss some advantages and some disadvantages of the particular page-table structure used in Windows 2000.
Answer: Each process has its own page directory that requires about 4megabytes of storage. Since it is a three level design, this means that there could be up to three page faults just accessing a virtual address. Shared memory adds one more level. The page faults can occur becauseWindows 2000 does not commit the requiredmemory (the 4megabytes) until necessary. Since each process has its own page directory, there is no way for processes to share virtual addresses. The prototype page-table entry adds a level of indirection but eliminates the update of multiple page-table entries for shared pages.

Paginación y Segmentación Combinadas

La segmentación, que es visible para el programador, tiene las ventajas antes citadas, incluida la capacidad de manejar estructuras de datos que puedan crecer, la modularidad y el soporte de la compartición y la protección. Para combinar las ventajas de ambas, algunos sistemas están equipados con hardware del procesador y software del sistema operativo que las permiten.

En un sistema con paginación y segmentación combinadas, el espacio de direcciones de un usuario se divide en varios segmentos según el criterio del programador. Cada segmento se vuelve a dividir en varias páginas de tamaño fijo, que tienen la misma longitud que un marco de memoria principal. Si el segmento tiene menor longitud que la página, el segmento ocupará sólo una página. Desde el punto de vista del programador, una dirección lógica también está formada por un número de segmento y un desplazamiento en el segmento. Desde el punto de vista del sistema, el desplazamiento del segmento se ve como un número de página dentro del segmento y un desplazamiento dentro de la página.

Asociada con cada proceso existe una tabla de segmentos y varias tablas de páginas, una por cada segmento del proceso. Cuando un proceso determinado está ejecutándose, un registro contendrá la dirección de comienzo de la tabla de segmentos para ese proceso. Dada una dirección virtual, el procesador emplea la parte de número de segmento como índice en la tabla de segmentos del proceso para encontrar la tabla de páginas de dicho segmento. Entonces, la parte de número de página de la dirección virtual se usará como índice en la tabla de páginas para localizar el número

de marco correspondiente. Este se combina con la parte de desplazamiento de la dirección virtual para generar la dirección real deseada.

Los bits de presencia y modificación no son necesarios, puesto que estos elementos se manejan a nivel de página. Pueden usarse otros bits de control para compartición y protección. La entrada de la tabla de páginas es, básicamente, la misma que se usa en un sistema de paginación pura. Cada número de página se convierte en el número de marco correspondiente si la página está presente en memoria. El bit de modificación indica si se necesita escribir la página a disco cuando se asigna el marco a otra página. Además, pueden haber otros bits de control para ocuparse de la protección y de otros aspectos de la gestión de memoria.

Why are segmentation and paging sometimes combined into one scheme?
Answer: Segmentation and paging are often combined in order to improve upon each other. Segmented paging is helpful when the page table becomes very large. A large contiguous section of the page table that is unused can be collapsed into a single segment table entry with a page-table address of zero. Paged segmentation handles the case of having very long segments that require a lot of time for allocation. By paging the segments, we reduce wasted memory due to external fragmentation as well as simplify the allocation.

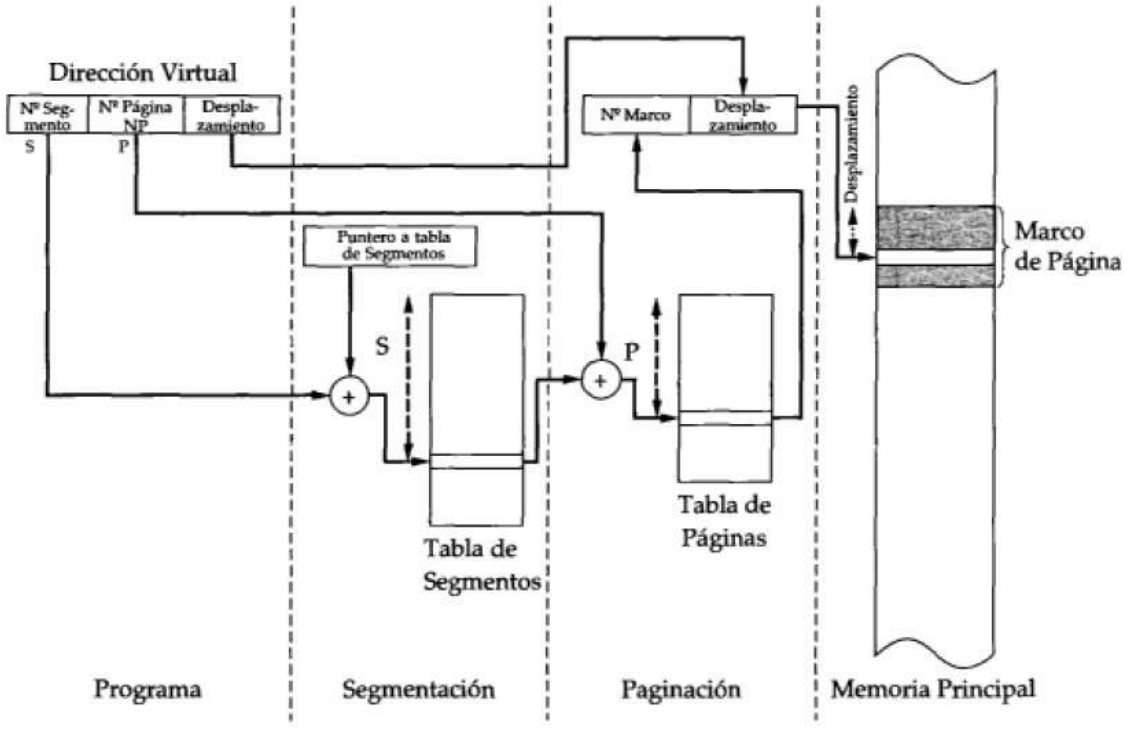


FIGURA 7.11 Traducción de direcciones en un sistema con segmentación y paginación

[tan]
ejemplos de paginación y segmentación combinadas:

Multics:

Si los segmentos son grandes, puede ser problemático, o incluso imposible, mantenerlos completos en la memoria principal. Esto nos lleva a la idea de paginarlos, de modo que sólo se consérvenlas

páginas que realmente se necesitan. Varios sistemas importantes han manejado segmentos paginados. En esta sección describiremos el primero: MULTICS. En la siguiente veremos uno más reciente: Pentium de Intel.

Cada programa MULTICS tiene una tabla de segmentos, con un descriptor por segmento. Puesto que puede haber más de un cuarto de millón de entradas en la tabla, la tabla de segmentos es en sí un segmento y se pagina. Un descriptor de segmento contiene una indicación de si el segmento está en la memoria principal o no. Si cualquier parte del segmento está en la memoria, se considera que el segmento está en la memoria, y su tabla de páginas estará en la memoria. Si el segmento está en la memoria, su descriptor contiene un apuntador de 18 bits a su tabla de páginas
Cada segmento es un espacio de direcciones virtual ordinario y se pagina del mismo modo que la memoria paginada no segmentada que se describió en una sección anterior de este capítulo. El tamaño de página normal es de 1024 palabras (aunque unos cuantos segmentos pequeños que utiliza MULTICS mismo no están paginados o se paginan en unidades de 64 palabras a fin de ahorrar memoria física).

Una dirección en MULTICS consta de dos partes: el segmento y la dirección dentro del segmento. La dirección dentro del segmento se subdivide en un número de página y una palabra dentro de la página, como se muestra en la Fig. 4-24. Cuando ocurre una referencia a la memoria, se lleva a cabo el siguiente algoritmo.

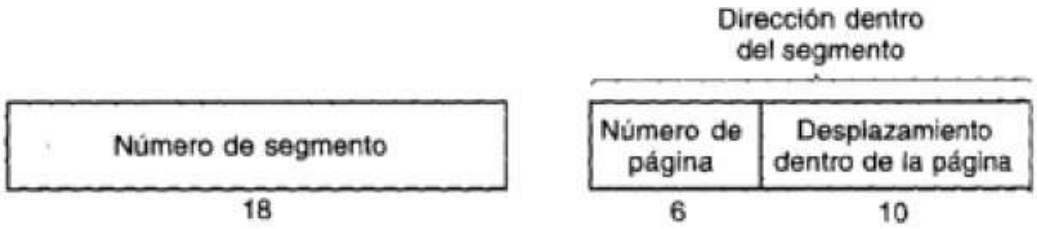


Figura 4-24. Dirección virtual MULTICS de 34 bits.

1. Se usa el número de segmento para encontrar el descriptor de segmento.
2. Se verifica si la tabla de páginas del segmento está en la memoria. Si es así, se le localiza; si no, ocurre una falla de segmento. Si hay una violación de la protección, ocurre una falla (trampa).
3. Se examina la entrada de tabla de páginas que corresponde a la página virtual solicitada. Si la página no está en la memoria, ocurre una falla de página; si está en la memoria, se extrae de la entrada de la tabla de páginas la dirección de principio de la página en la memoria principal.
4. Se suma la distancia al origen de la página para obtener la dirección en la memoria principal donde se encuentra la palabra.
5. Finalmente se efectúa la lectura o el almacenamiento.

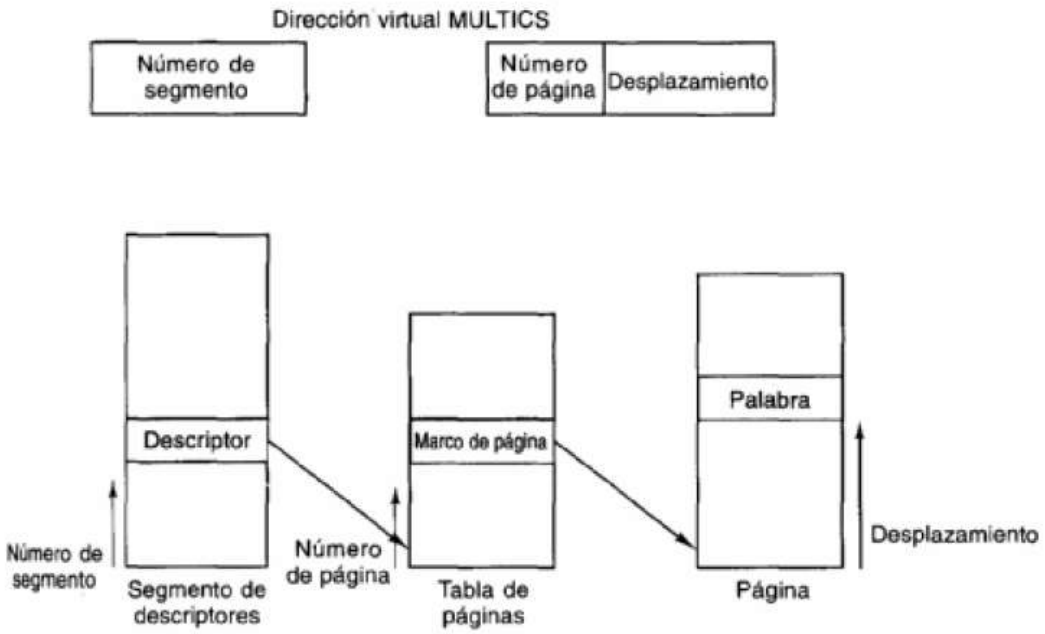


Figura 4-25. Conversión de una dirección MULTICS de dos partes en una dirección de memoria principal.

19.1.4 Segmentación:

Consecuencias de la Memoria Virtual

La segmentación permite al programador contemplar la memoria como si constara de varios espacios de direcciones o segmentos. Con memoria virtual, el programador no necesita preocuparse de las limitaciones de memoria impuestas por la memoria principal. Los segmentos pueden ser de distintos tamaños, incluso de forma dinámica. Las referencias a memoria constan de una dirección de la forma (número de segmento, desplazamiento).

Esta organización ofrece al programador varias ventajas sobre un espacio de direcciones no segmentado:

1. Simplifica el manejo de estructuras de datos crecientes. Si el programador no conoce *a priori* cuan larga puede llegar a ser una estructura de datos determinada, es necesario suponerlo a menos que se permitan tamaños de segmento dinámicos. Con memoria virtual segmentada, a cada estructura de datos se le puede asignar a su propio segmento y el sistema operativo expandirá o reducirá el segmento cuando se necesite. Si un segmento necesita expandirse en memoria y no dispone de suficiente sitio, el sistema operativo puede mover el segmento a un área mayor de la memoria principal, si la hay disponible, o descargarlo. En este último caso, el segmento agrandado será devuelto a memoria en la siguiente ocasión.

2. Permite modificar y recompilar los programas independientemente, sin que sea necesario recompilar o volver a montar el conjunto de programas por completo. Esto se puede llevar nuevamente a cabo gracias al uso de varios segmentos.
3. Se presta a la compartición entre procesos. Un programador puede situar un programa de utilidades o una tabla de datos en un segmento que pueda ser referenciado por otros procesos.
4. Se presta a la protección. Puesto que un segmento puede ser construido para albergar un conjunto de procedimientos y datos bien definido, el programador o el administrador del sistema podrá asignar los permisos de acceso de la forma adecuada.

19.1.5 Organización

En el estudio de la segmentación simple, se llegó a la conclusión de que cada **proceso tiene su propia tabla de segmentos** y que, cuando todos los segmentos se encuentran en memoria principal, la tabla de segmentos del proceso se crea y carga en memoria. Cada entrada de la tabla de segmentos contiene la dirección de comienzo del segmento correspondiente en memoria principal, así como su longitud. La misma estructura, una tabla de segmentos, se necesitara al hablar de un esquema de memoria virtual basado en segmentación. **Nuevamente, es normal asociar una única tabla de segmentos a cada proceso.** En este caso, sin embargo, las entradas de la tabla de segmentos pasan a ser más complejas (figura 7.2b**). Puesto que sólo algunos de los segmentos de un proceso estarán en memoria principal, se necesita un bit en cada entrada de la tabla de segmentos para indicar si el segmento correspondiente está presente en memoria principal**. Si el bit indica que el segmento está en memoria, la entrada incluye también la dirección de comienzo y la longitud del segmento.

Otro bit de control necesario en la entrada de la tabla de segmentos es un **bit de modificación** que indique si el contenido del segmento correspondiente ha sido modificado desde que se cargó por última vez en memoria principal. **Si no ha habido cambios, no será necesario escribir a disco el segmento cuando llegue el momento de reemplazarlo en el marco que ocupa actualmente**. Puede haber también **otros bits** de control. Por ejemplo, si se gestiona la **protección** o la **compartición** a nivel del segmento, se necesitarán bits con tal propósito. Más adelante se verán varios ejemplos de entradas de tablas de segmentos.

Así pues, el mecanismo básico para leer una palabra de memoria supone la traducción de una dirección virtual o lógica, formada por un número de segmento y un desplazamiento, a una dirección física, mediante una tabla de segmentos. **Puesto que la tabla de segmentos tiene longitud variable, en función del tamaño del proceso, no es posible suponer que quepa en registros.** En su lugar, **debe estar en memoria principal** para que sea accesible. La figura 7.10 sugiere una implementación en hardware de este esquema. Cuando un proceso determinado está ejecutando, la dirección de comienzo de la tabla de segmentos de este proceso se guardará en un registro. El número de segmento de la dirección virtual se emplea como índice de la tabla para buscar la dirección de memoria principal correspondiente al comienzo del segmento. Esta se añade a la parte de desplazamiento de la dirección virtual para generar la dirección real deseada.

19.1.6 Protección y Compartición:

La segmentación se presta a la implementacióni de políticas de protección y compartición. Puesto que cada entrada de la tabla de segmentos incluye la longitud, además de la dirección base, un programa no podrá acceder por descuido a una posición de memoria principal más allá de los límites de un segmento. Para conseguir la compartición, es posible que un segmento se referencie

en las tablas de segmentos de más de un proceso. Este mismo mecanismo es, por supuesto, válido en un sistema de paginación. Sin embargo, es este caso, la estructura de páginas de programas y datos no es visible al programador, haciendo que la especificación de los requisitos de protección y seguridad sea más difícil. La figura 7.12 indica el tipo de relaciones de protección que se pueden definir entre segmentos. Se pueden ofrecer también mecanismos más sofisticados. Un esquema habitual consiste en usar una estructura de anillo de protección como la que se explicó en el capítulo 3. Con este esquema, los anillos más interiores o con números menores gozan de mayores privilegios que los anillos externos o con números mayores. La figura 7.13 ilustra los tipos de protección que pueden aplicarse en cada sistema. Normalmente, el anillo O está reservado para las funciones del núcleo del sistema operativo y las aplicaciones están situadas en un nivel más alto. Algunas utilidades o servicios del sistema operativo pueden ocupar un anillo intermedio. Los principios básicos del sistema de anillos son los siguientes: 1. Un programa puede acceder sólo a datos que estén en el mismo anillo o en un anillo de menor privilegio. 2. Un programa puede hacer llamadas a servicios que residan en el mismo anillo o en anillos más privilegiados.

Consideración	Paginación	Segmentación
¿El programador necesita estar consciente de que se está usando esta técnica?	No	Sí
¿Cuántos espacios de direcciones lineales hay?	1	Muchos
¿El espacio de direcciones total puede exceder el tamaño de la memoria física?	Sí	Sí
¿Se pueden distinguir los procedimientos de los datos y protegerse de forma independiente?	No	Sí
¿Se pueden manejar fácilmente tablas cuyo tamaño fluctúa?	No	Sí
¿Se facilita la compartición de procedimientos entre usuarios?	No	Sí
¿Por qué se inventó esta técnica?	Para tener un espacio de direcciones lineal grande sin tener que adquirir más memoria física.	Para poder dividir los programas y los datos en espacios de direcciones lógicamente independientes y facilitar la compartición y la protección.

Figura 4-21. Comparación de paginación y segmentación.

19.1.7 SOFTWARE DEL SISTEMA OPERATIVO

El diseño del gestor de memoria en un sistema operativo depende de tres áreas fundamentales de decisión:

• Si se emplean o no técnicas de memoria virtual

• El uso de paginación, segmentación o ambas
• Los algoritmos empleados para los diversos problemas de la gestión de memoria

Las decisiones tomadas en las dos primeras áreas dependen de la plataforma de hardware disponible. De este modo, las primeras implementaciones de UNIX no ofrecían memoria virtual, porque los procesadores en los que ejecutaba el sistema operativo no ofrecían paginación ni segmentación.
Las decisiones del tercer punto (los algoritmos) entran en el dominio del software del sistema operativo y son el objeto de esta sección.

4.5.1 El modelo de conjunto de trabajo

En la forma más pura de paginación, los procesos se inician con ninguna de sus páginas en la memoria.
Tan pronto como la CPU trata de obtener la primera instrucción, detecta una falla página que hace que el sistema operativo traiga a la memoria la página que contiene dicha instrucción. Por lo regular, pronto ocurren otras fallas de página al necesitarse variables globales y la pila. Después de un tiempo, el proceso tiene la mayor parte de las páginas que necesita y se dedica a ejecutarse con relativamente pocas fallas de página. Tal estrategia se denomina **paginación por demanda** porque las páginas se cargan sólo cuando se piden, no por adelantado.
Desde luego, es fácil escribir un programa de prueba que lea sistemáticamente todas páginas de un espacio de direcciones grande, causando tantas fallas de página que no habrá suficiente memoria para contenerlas todas. Por fortuna, la mayor parte de los procesos no funcionan así; exhiben una **localidad de referencia**, lo que significa que durante cualquier fase de ejecución, el proceso sólo hace referencia a una fracción relativamente pequeña de sus páginas. Cada pasada de un compilador de múltiples pasadas, por ejemplo, sólo hace referencia a fracción de todas las páginas, que es diferente en cada pasada.
El conjunto de páginas que un proceso está usando actualmente es su **conjunto de trabajo** (Denning, 1968a; Denning, 1980). Si todo el conjunto de trabajo está en la memoria, el proceso ejecutará sin causar muchas fallas hasta que pase a otra fase de su ejecución (p. ej., la siguiente pasada de un compilador). Si la memoria disponible es demasiado pequeña para contener todo conjunto de trabajo, el proceso causará muchas fallas de página y se ejecutará lentamente, ya que la ejecución de una Instrucción normalmente toma unos cuantos nanosegundos, y la lectura; una página de disco suele tomar decenas de milisegundos. Si ejecuta sólo una o dos instrucción cada 20 milisegundos, el proceso tardará muchísimo en terminar. Se dice que un programa q causa fallas de página repetidamente después de unas cuantas instrucciones se está "sacudiendo (en inglés, thrashing, que es el término que utilizaremos) (Denning,1968b). En un sistema de tiempo compartido, los procesos a menudo se transfieren al disco (esto es, todas sus páginas se eliminan de la memoria) para que otro proceso tenga oportunidad de usar la CPU. Surge la pregunta de qué hacer cuando se vuelve a traer un proceso a la memoria. Técnicamente, no hay que hacer nada. El proceso simplemente causará fallas de página hasta que haya cargado su conjunto de trabajo. El problema es que tener 20, 50 o incluso 100 fallas de página cada vez que se carga un proceso hace lento el funcionamiento y además desperdicia mucho tiempo de CPU, pues el sistema operativo requiere unos cuantos milisegundos de tiempo de CPU para procesar una falla de página. Por ello, muchos sistemas de paginación tratan de seguir la pista al conjunto de trabajo de cada proceso y se aseguran de que esté en la memoria antes de dejar que el proceso se ejecute. Este enfoque se denomina **modelo de conjunto de trabajo** (Denning, 1970) y está diseñado para reducir
considerablemente la tasa de fallas de página. La carga de las páginas antes de dejar que los procesos se ejecuten también se denomina **prepaginación**.
A fin de implementar el modelo de conjunto de trabajo, el sistema operativo tiene que saber cuáles páginas están en el conjunto de trabajo. Una forma de seguir la pista a esta información es usar el algoritmo de maduración que acabamos de explicar. Cualquier página que contenga un bit 1 entre los n bits de orden alto del contador se considera miembro del conjunto de trabajo. Si no se ha

hecho referencia a una página en n tics de reloj consecutivos, se omite del conjunto de trabajo. El parámetro n se debe determinar experimentalmente para cada sistema, pero por lo regular el rendimiento del sistema no es muy sensible al valor exacto.
La información relativa al conjunto de trabajo puede servir para mejorar el rendimiento del algoritmo del reloj. Normalmente, cuando la manecilla apunta a una página cuyo bit R es O, esa página se desaloja. La mejora consiste en verificar si esa página forma parte del conjunto de trabajo del proceso en curso. Si lo es, se le perdona la vida. Este algoritmo se conoce como **wsclock**.

19.1.8 Reemplazo de Páginas

En los esquemas vistos la frecuencia de fallos de página no han representado un problema grave. Pensemos en que un proceso que requiere 10 páginas, generalmente necesita 5 de ellas cargadas. Con este esquema ahorramos la E/S de traer las 10 páginas a memoria, además se puede ampliar la multiprogramación ejecutando el doble de procesos (ya que cargo la mitad de las páginas).

Si uno de estos procesos que usa 5 paginas produce un fallo de pagina (requiere de una sexta pagina), ocurre una trampa y se transfiere el control al SO. No quedan marcos libres, toda la memoria está ocupada. Al llegar a este estado, el SO tiene varias opciones:

- **Terminar el proceso de usuario.** Esta opción no es muy aceptable debido a que el SO hace paginación por demanda para aumentar la productividad del computador, y además este esquema debe ser transparente para el usuario.
- **Intercambiar a disco un proceso.** Haciendo esto se liberan algunos marcos, pero se reduce el nivel de multiprogramación. Esta opción es bastante buena pero hay otra más interesante.
- **Reemplazo de páginas.** Si no hay marcos libres se busca uno que no se este usando y lo liberamos.

La liberación de un marco es posible hacerla escribiendo su contenido en el espacio de intercambio y modificando la tabla de páginas (y las demás tablas) de modo que indiquen que la página ya no esta en la memoria. Ahora el marco esta libre y puede contener la pagina por la cual ocurrió un fallo.

Entonces, debemos modificar la rutina de servicio de fallos de página (capítulo 9.2) de la siguiente forma:

```
Si hay un marco libre
        Se usa;
Sino
{
        Se usa un algoritmo de reemplazo de paginas para escoger el marco victima;
        Escribir la pagina victima en el disco y modificar las tablas de paginas y
marcos;
        Leer la página deseada y colocarla en al marco liberado, modificar tablas;
}
Fin Si
```

Si no hay un marco libre, se requieren 2 transferencias de páginas (una hacia fuera y una hacia adentro), duplicando el tiempo de servicio de fallo de página. Este gasto extra puede ser solucionado empleando un *bit de modificación* o *bit sucio*.

Cada página o marco puede tener en hardware un bit asociado. El cual es encendido si se escribe una palabra o byte en la página (indicando que se modifico). De esta forma evitamos la escritura de la pagina victima si no ha sido modificada.

El reemplazo de páginas es básico para la paginación por demanda, pues hace la total separación entre la memoria lógica y física, y ofrece a los programadores una memoria virtual muy grande con una memoria física muy pequeña.

Para implementar la paginación por demanda, entonces debemos resolver 2 problemas:

- **Un algoritmo de asignación de marcos.** Es necesario decidir cuantos marcos se asignan a cada uno.
- **Un algoritmo de reemplazo de páginas.** Hay que seleccionar que página se reemplaza cuando se necesario.

  Estos algoritmos deben ser bien diseñados para ganar un buen desempeño del computador.

19.1.9 Algoritmo de Reemplazo de Páginas

Hay muchos, y es probable que cada SO tenga su propio esquema de reemplazos. Lo que queremos es el algoritmo con *frecuencia de fallos de página* más baja.

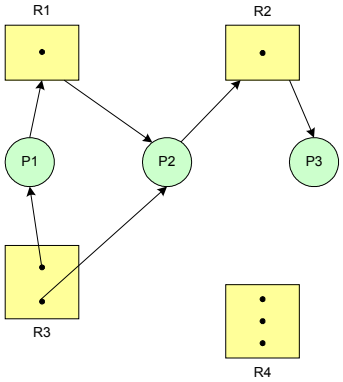
La evaluación de los algoritmos se hace ejecutándolo con una serie específica de referencias a la memoria (denominada *serie de referencias*) y calculando el número de fallos de página. Además se debe saber el número de marcos disponibles en el sistema (cuantos más hallan menos fallos ocurrirán).

19.1.9.1 Algoritmo FIFO

Es el más sencillo. Este algoritmo asocia a cada página el instante en que se trajo a la memoria. Cuando es necesario reemplazar una página se escoge la más vieja. Esto se puede implementar con una cola simple.

Su desempeño no es siempre bueno, ya que la pagina más vieja puede contener una variable que se esta consultando constantemente.

Un problema importante de este algoritmo se conoce como *anomalía de Belady*. Consiste en que al aumentar la cantidad de marcos pueden aumentar la cantidad de fallos para algunas series de referencias.



19.1.9.2 Algoritmo óptimo

Este algoritmo reemplaza la página que no se **usará** durante más tiempo, asegurando la frecuencia de fallos más baja para un número fijo de marcos.

El problema es que es difícil de implementar porque requiere un conocimiento futuro de la serie de referencias.

19.1.9.3 Algoritmo LRU

Least recently used, es el algoritmo que reemplaza la página que no se ha usado por más tiempo. Asocia a cada página el instante en que se uso por ultima vez, y así selecciona la que hace más tiempo que no se usa.

Esta política se usa mucho y se considera excelente, el problema es como implementarla, ya que requiere ayuda sustancial del hardware.

Hay 2 implementaciones posibles:

- **Contadores:** Asociamos a cada entrada de la tabla de páginas el campo de "tiempo de uso" y añadimos al CPU un reloj lógico o contador. El reloj se incrementa en cada referencia a la memoria. Cada vez que se hace referencia a una página, el contenido del reloj se copia en el campo de tiempo de uso. Luego hay que hacer una búsqueda para encontrar la página. Otro problema a resolver es el overflow del reloj.
- **Pila:** La estrategia es mantener una pila de números de página. Cuando se hace referencia a una se saca de la pila y se coloca arriba. Así en el tope esta la más reciente y en la base la LRU. La pila se debe implementar como una lista doblemente enlazada. Cada actualización es un poco más costosa, pero ya no es necesario buscar para reemplazar. Este enfoque es el más apropiado para implementaciones en software.

19.1.9.4 Algoritmos de Aproximación a LRU

Pocos sistemas de computación ofrecen suficiente soporte de hardware para el verdadero reemplazo de paginas LRU. Los que no cuentan con apoyo de hardware no pueden hacer otra cosa que usar el FIFO. Los que cuentan con un bit que el hardware enciende cada vez que se accede a una página, (permitiendo saber que páginas se usaron y cuales no), pueden utilizar algoritmos que se aproximan a LRU.

19.1.9.4.1 Algoritmo con Bits de Referencias Adicionales

Mantenido un byte histórico para cada página de una tabla que esta en memoria. A intervalos regulares una interrupción ejecuta una rutina del SO que desplaza un bit a la derecha en el byte histórico y pone el bit de referencia de la página en el bit más significativo (del byte histórico). Así la página con número más bajo será la elegida, y como el número puede no ser único se puede eliminar diferencias con FIFO u otro algoritmo.

19.1.9.4.2 Algoritmo de Segunda Oportunidad

El algoritmo es igual a FIFO, pero cuando se va a elegir una página se mira el bit de referencia, si es 0 se reemplaza, sino se le da una segunda oportunidad (y se pone el bit en 0), mirando a continuación la siguiente en orden FIFO.

La implementación es generalmente mediante una cola circular.

19.1.9.4.3 Algoritmo de Segunda Oportunidad Mejorado

Consiste en considerar tanto el bit de referencia como el bit de modificación como un par ordenado. Obteniendo los 4 siguientes casos:

- **(0, 0).** Ni se uso ni se modifico, es la mejor para reemplazar.
- **(0, 1).** No se uso pero se modifico, hay que escribirla en disco antes de reemplazarla por lo cual no es tan buena.
- **(1, 0).** Se uso pero no se modifico, es posible que se use pronto.
- **(1, 1).** Se uso y se modifico, se va a usar probablemente pronto y será necesario escribirla antes de reemplazarla.

Este esquema es utilizado por Macintosh

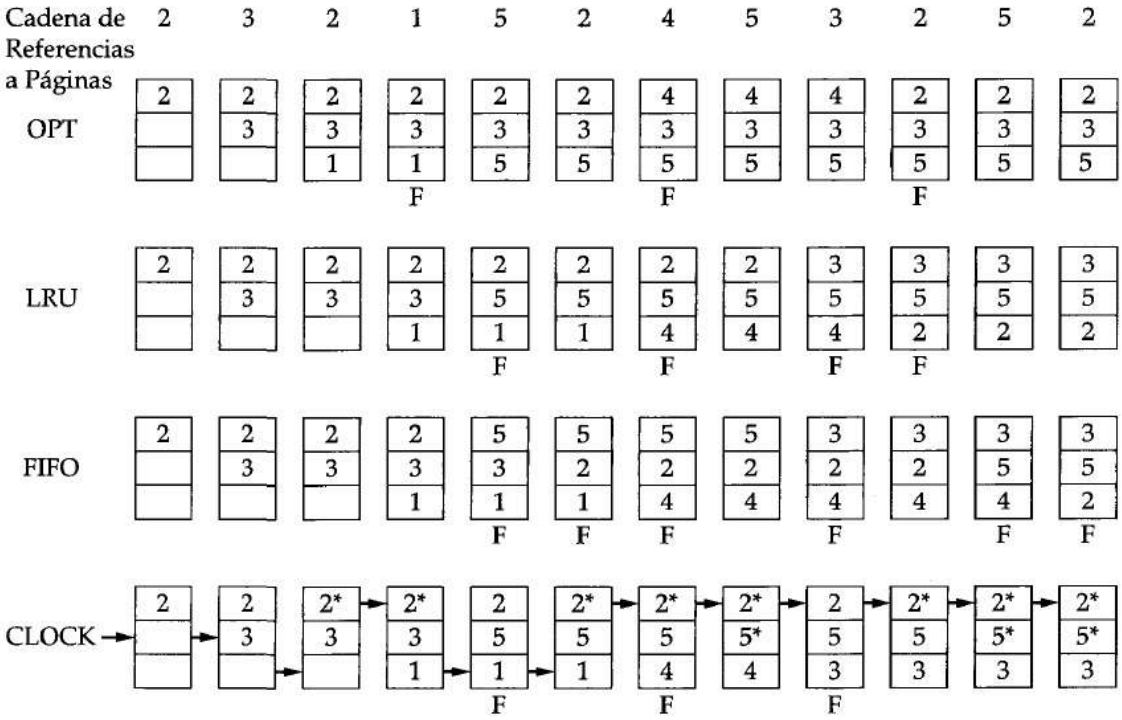


FIGURA 7.14 Comportamiento de cuatro algoritmos de reemplazo de páginas (basado en [HAYE88])

19.1.9.5 Algoritmos de Conteo

Manteniendo un contador de número de referencias que se han hecho a cada página se puede hacer:

- **Algoritmo LFU.** Se elige el menos frecuentemente usado (*least frecuently used)*, el de cuenta más baja. No es eficiente en el caso de páginas que se usaron mucho al comienzo del proceso, y luego no se volvieron a usar (tienen cuenta alta).
- **Algoritmo MFU.** Se elige la más frecuentemente usada. Se basa en el argumento de que la última página traída a memoria, es la que tiene cuenta más baja.

Estos algoritmos no son comunes y poco se aproximan al ÓPTIMO.

19.1.9.6 Algoritmo de Colocación de Páginas en Buffers

Muchos sistemas mantienen una *reserva* de marcos libre. Cuando ocurre un fallo de pagina se escoge un marco víctima igual que antes, pero, la página deseada se coloca en un marco libre de la reserva antes de escribir la víctima en disco. Este procedimiento permite al proceso reiniciarse lo más pronto posible, sin esperar que la victima se escriba en el disco. Cuando termino de escribir la víctima, añado su marco a la reserva de marcos libres.

También se puede usar esta idea, para ir escribiendo en disco las páginas modificadas, así, cuando son elegidas para ser reemplazadas están “límpias”.

Otra forma, es manteniendo una reserva de marcos libres, pero recordando que página estaba en cada uno, así, si fue escrita a disco, reemplazada, y la volvemos a necesitar, todavía se puede encontrar en el marco libre, sin ser sobrescrita.

19.1.10Asignación de Marcos

Uno de los grandes problemas es como saber cuantos marcos hay que darle a cada proceso, sobretodo cuando el sistema es multiprogramado (hay varios procesos en memoria a la vez).

19.1.10.1Numero Mínimo de Marcos

El número máximo esta definido por la cantidad de memoria física disponible. Como vimos, cuantos menos marcos asignemos la frecuencia de fallos de página aumenta, por eso hay un número mínimo de marcos que es preciso asignar y esta definido por la arquitectura del conjunto de instrucciones. Es necesario tener suficientes marcos para contener todas las páginas a las que una sola instrucción puede hacer referencia.

La situación más grave ocurre en arquitecturas que permiten múltiples niveles de indirección, por lo tanto, una página podría tener referencia a otra, y ésta a otra, y así

sucesivamente, necesitando tener cargadas todas las páginas en memoria para poder ejecutar una simple instrucción.

Por esta causa, se limita el nivel de indirección asignando como máximo, la cantidad de marcos disponibles en la memoria física.

19.1.10.2Algoritmos de Asignación

- **Asignación Equitativa.** Se dividen los m marcos entre los n procesos, dando a cada uno una porción equitativa de la cantidad de marcos.
- **Asignación Proporcional.** Se le asigna memoria disponible a cada proceso según su tamaño.

Tanto en la equitativa como en la proporcional la asignación a cada proceso podría variar según el nivel de multiprogramación (cuantos más procesos haya en memoria, menor es la cantidad de marcos asignados a cada uno). Además, en ambos casos se trata igual a un proceso de alta prioridad que a un proceso de baja prioridad. Por esto, otras estrategias ofrecen marcos dependiendo del nivel de prioridad, o según una combinación de tamaño y prioridad del proceso.

19.1.10.3Asignación Global o Local

Podemos clasificar los algoritmos de reemplazo de páginas en dos categorías:

- **Reemplazo Global.** Permite seleccionar la víctima de todos los marcos disponibles, incluyendo los de otros procesos. Un problema es que un proceso no puede controlar su propia frecuencia de fallos de página. Es el más utilizado, ya que aumenta el rendimiento del sistema.
- **Reemplazo Local.** Selecciona la victima solo de los marcos que le fueron asignados. Un problema es que un proceso se puede entorpecer al no poder aprovechar otras páginas que podrían tener muy poco uso.

19.1.11Hiperpaginación (Trashing)

Un proceso que no tiene suficientes marcos para su cantidad de *páginas activas* (número de páginas que siempre esta usando), provocará un fallo de página a cada instante, por lo que comenzará a estar en un estado de frenética paginación, pasando más tiempo paginando que ejecutando, y esto se denomina *hiperpaginación.*

19.1.11.1Causa de la Hiperpaginación

La hiperpaginación trae serios problemas de desempeño. El SO supervisa el aprovechamiento del CPU y cuando detecta que es bajo, introduce un nuevo proceso para aumentar la multiprogramación. En un sistema con reemplazo global, este proceso puede generar un fallo si tiene pocas páginas y al hacerlo le quita páginas a otros procesos, los otros procesos hacen lo mismo, y todos comienzan a hiperpaginar,

terminando con un aprovechamiento del CPU peor al que había antes. Al bajar el aprovechamiento, el SO introduce otro proceso, y así sucesivamente hasta desplomar el rendimiento del sistema.

Al agregar procesos, el rendimiento aumenta, hasta llegar a un punto máximo de procesos concurrentes. Si se siguen agregando procesos, el rendimiento cae abruptamente. A partir de ese punto, para aumentar el rendimiento, hay que disminuir la cantidad de procesos concurrentes.

Este efecto se puede limitar utilizando un algoritmo de reemplazo local (o por prioridad). Pero para evitar la hiperpaginación hay que darle a cada proceso los marcos que necesita.

19.1.11.2Modelo de Conjunto de Trabajo

Se basa en el supuesto de localidad. Este modelo emplea un parámetro D para definir la *ventana del conjunto de trabajo*. De lo que se trata es de examinar las D referencias a páginas más recientes, y esas D páginas es el *conjunto de trabajo*. Si una página esta en uso activo estará en el conjunto de trabajo; si ya no se esta usando saldrá del conjunto, D unidades de tiempo después de su ultima referencia. Así, el conjunto de trabajo es una aproximación de localidad del programa.

El sistema operativo vigila el conjunto de trabajo de cada proceso y le asigna un número de marcos igual a su tamaño. Si hay marcos libres, puede iniciar otro proceso. Si la suma de los tamaños de los conjuntos de trabajo aumenta hasta exceder el numero total, el SO suspenderá un proceso (y se iniciará más tarde).

Esto evita la hiperpaginación y mantiene la multiprogramación lo más alta posible. El problema es que es difícil seguir la pista a los conjuntos de trabajos.

19.1.11.3Frecuencia de Fallos de Pagina

Podemos establecer límites superiores e inferiores para la frecuencia de fallos de páginas deseada. Si la frecuencia de fallos real excede el límite superior, se le da al proceso otro marco y si baja del inferior se le quita uno.

Evitamos así la hiperpaginación, pero en el caso de que la frecuencia de fallos de página aumente y no haya marcos disponibles, deberemos seleccionar un proceso y suspenderlo.

19.1.12Otras Consideraciones

19.1.12.1Prepaginación

Es un intento por evitar el alto nivel de fallos de página al iniciar el proceso. La estrategia consiste en traer a memoria en una sola operación todas las páginas que se necesiten, para ello se debe guardar junto con cada proceso, una lista de las páginas de su *conjunto de trabajo*. Esto puede ser ventajoso en algunos casos. Hay que considerar

si el costo de la prepaginación es menor que el de atender los fallos de página correspondientes.

19.1.12.2Tamaño de página

La memoria se aprovecha mejor con páginas pequeñas, pero como cada proceso activo mantiene su tabla de páginas es preferible páginas grandes (para tener una tabla más pequeña).

Otro problema a considerar es el tiempo que toma leer o escribir una página, y este es un argumento a favor de las páginas grandes.

Con páginas pequeñas, tenemos mejor *definición*, solo traemos a memoria lo realmente necesario, con páginas más grandes, debemos transferir todo lo que contenga la página, sea necesario o no. Pero cuanto más grandes sean las páginas, menor es la cantidad de fallos de página (que producen gastos extra).

La tendencia histórica es a páginas más grandes, pero no hay una postura definida.

19.1.12.3Tablas de Páginas Invertidas

El propósito es reducir la cantidad de memoria física necesaria para las traducciones de direcciones virtuales a físicas. El problema es que no contienen la información necesaria para trabajar con paginación por demanda, necesitando una tabla de páginas externa para cada proceso, que contengan esa información.

19.1.12.4Estructura del Programa

La paginación por demanda es transparente al programa de usuario. Sin embargo se puede mejorar el desempeño del sistema si se conoce.

Ver ejemplo página 325 del Silberschatz (útil para el práctico).

19.1.12.5Interbloqueo de E/S

A veces es necesario permitir que se *bloqueen* o fijen algunas páginas en memoria. Esto implica que no se reemplacen páginas que están esperando una operación de E/S, o páginas que son utilizadas como buffers para el mismo fin (E/S). La forma de implementarlo es asociando un bit de bloqueo a cada marco.

Otro motivo de bloqueo, es cuando un proceso de baja prioridad, trae a memoria una página, y antes de usarla es desplazado por un proceso de mayor prioridad. El proceso de baja prioridad va a la cola de procesos listos, y permanece allí por bastante tiempo (debido a su baja prioridad). Los marcos de memoria van a ir siendo reemplazados, hasta que se reemplaza el marco del proceso de baja prioridad.

Estamos desperdiciando la labor invertida en traer la página del proceso de baja prioridad y para impedirlo, podemos bloquearla.

El bloqueo es peligroso, si el bit se enciende pero nunca se apaga (por algún fallo del sistema), ya que el marco queda inutilizado.

19.1.12.6Procesamiento en Tiempo Real

La memoria virtual trae retardos a los procesos, por lo cual no es posible utilizarla en sistemas de tiempo real.

19.1.13Segmentación por Demanda

La paginación por demanda se considera el sistema de memoria virtual más eficiente, pero se requiere de mucho hardware para implementarlo. Si falta el hardware una opción posible es la *segmentación por demanda*.

El funcionamiento es muy similar a la paginación por demanda, al igual que sus algoritmos y soluciones a problemas. Pero, el gasto es mayor ya que los segmento son de tamaño variable y esto complica los algoritmos.

20 Bloqueos mutuos (Deadlock)

Cuando un proceso A solicita un recurso que está ocupado por otro proceso B, pasa a un estado de espera. Si B también está en estado de espera, esperando un recurso que tiene A, los dos procesos están en deadlock.

20.1.1 Modelo del sistema

Un sistema consiste en un número finito de recursos que deben distribuirse entre varios procesos que compiten. Los recursos se dividen en varios tipos, cada uno de los cuales consiste en cierta cantidad de ejemplares idénticos (unidades de recursos). Si un proceso solicita un ejemplar de un tipo de recurso, la asignación de cualquier ejemplar del tipo, satisface la solicitud.

Un proceso solo puede utilizar un recurso, siguiendo esta secuencia:

1. **Solicitud:** Si la solicitud no puede atenderse de inmediato, el proceso debe esperar.
2. **Uso:** El proceso opera con el recurso.
3. **Liberación:** El proceso libera el recurso.

La solicitud y liberación, son llamadas al SO. Una tabla del sistema registra si cada recurso está libre o asignado. Si un proceso solicita un recurso ocupado, se lo coloca en la cola de procesos que esperan ese recurso.

Los recursos pueden ser físicos (impresoras, memoria, ciclos de CPU), o lógicos (archivos, semáforos, monitores).

20.1.2 7.2 Caracterización de bloqueos mutuos

En un bloque mutuo, los procesos nunca terminan su ejecución y los recursos del SO quedan acaparados.

20.1.2.1 Condiciones necesarias

Puede llegar a ocurrir bloqueo mutuo si se cumplen simultáneamente las cuatro condiciones siguientes:

- **Mutua exclusión:** Al menos un recurso debe adquirirse de forma no compartida, o sea, solo puede usarlo un proceso a la vez. Si otro proceso lo solicita, debe esperar.
- **Retener y esperar:** Debe existir un proceso que haya adquirido al menos un recurso y esté esperando para adquirir recursos adicionales, que ya fueron otorgados a otros procesos.
- **No expropiación:** Los recursos no se pueden arrebatar, es decir, la liberación es voluntaria por parte del proceso que adquirió el recurso.
- **Espera circular:** Debe existir un conjunto $\{P_0, P_1, \dots, P_n\}$ de procesos en espera tal que P_0 espera un recurso adquirido por P_1, P_1 espera uno adquirido por P_2, … , P_n espera uno adquirido por P_0.

20.1.2.2 Grafo de asignación de recursos

Los bloqueos mutuos se pueden describir con mayor precisión en términos del *grafo de asignación de recursos del sistema*. Los vértices son P y R (procesos y recursos). Una arista de P a R indica que el proceso espera el recurso, y una de R a P, indica que se asignó el recurso al proceso.

P se identifica con un círculo, R con un cuadrado, y puntos adentro, donde cada punto es una unidad de recurso.

Casos:

- **No contiene ciclos:** No existe bloqueo mutuo.
- **Si contiene ciclos:**
  - Si los recursos solo tienen un único ejemplar, hay bloqueo mutuo
  - Si hay varios ejemplares, no necesariamente hay bloqueo.

20.1.3 Métodos para manejar bloqueos mutuos

- Usar un protocolo que asegure que el SO nunca llegará a deadlock
- El sistema entra en un bloqueo mutuo y se recupera
- Desentenderse del problema y hacer como si nunca ocurrieran bloqueos (es lo que hacen la mayor parte de los SO)

Prevención de bloqueos mutuos: Métodos que garantizan que no se cumpla por lo menos una de las condiciones necesarias.

Evitación de bloqueos mutuos: Requiere proporcionar al SO información anticipada acerca de los recursos que el proceso va a solicitar y usar. Con esa información el SO decide si el proceso va a esperar o no.

Cuando el SO no cuenta con ninguno de los dos algoritmos anteriores, pueden ocurrir bloqueos, los cuales deben ser detectados y revertidos, de otra forma, el SO se deteriora hasta dejar de funcionar.

20.1.4 Prevención de bloqueos mutuos

Podemos tratar de que no se cumpla alguna de las condiciones necesarias para bloqueos mutuos.

20.1.4.1 Mutua exclusión

Se debe cumplir mutua exclusión para recursos no compartibles. Por ejemplo, dos o más procesos no pueden compartir una impresora, pero si pueden compartir un archivo de solo lectura.

20.1.4.2 Retener y esperar

Siempre que un proceso solicite un recurso, no debe retener otro. Una forma es que el proceso solicite y reciba todos los recursos antes de iniciar su ejecución. Otra forma es que pueda solicitar recursos solo cuando no tiene ninguno, para solicitar nuevos, suelto los que tengo.

Por ejemplo si tengo un proceso que copia datos de una cinta al disco, y luego imprime los resultados en una impresora, con el primer método, solicito la impresora al comienzo, para solo usarla al final, y mientras tanto, nadie la puede usar. Con el segundo método, solicito solo la cinta y el archivo de disco, y cuando termino con los dos, solicito nuevamente el archivo de disco y la impresora.

Las desventajas de los dos protocolos son que si mi proceso utiliza recursos que son muy solicitados, podría esperar indefinidamente a que todos se liberen para usarlos *(inanición)*.

20.1.4.3 No expropiación

Si un proceso tiene recursos solicitados, y solicita otro, que no se le puede asignar de inmediato, entonces todos los recursos que tiene se liberarán implícitamente. Es aplicable solo a recursos cuyo estado es fácil de guardar.

20.1.4.4 Espera circular

Se impone un ordenamiento total de los recursos y se exige que cada proceso los solicite en orden creciente de enumeración. Cuando se requieren varios ejemplares de un mismo recurso, se debe emitir una sola solicitud que los incluya a todos. Para pedir recursos con número mayor, debo liberar los que ya tengo.

20.1.5 Evitación de bloqueos mutuos

La prevención de bloqueos, disminuye el rendimiento del sistema. Se debe pedir información adicional para saber la forma en que se solicitarán los recursos, y el SO debe considerar los recursos disponibles, los ya asignados a cada proceso y las solicitudes y liberaciones futuras de cada proceso para decidir si la solicitud se puede satisfacer o debe esperar.

Una forma es saber el número máximo de recursos de cada tipo que se podría necesitar. El *estado de asignación* de los recursos está definido por el número de recursos disponible y asignado, y las demandas máximas.

20.1.5.1 Estado seguro

Un estado es *seguro*, si el sistema puede asignar recursos a cada proceso en algún orden, evitando bloqueos. Es seguro si existe una secuencia segura de procesos, si no existe, el estado del sistema es *inseguro*, pudiendo derivar en bloqueo mutuo.

La idea es asegurar que el sistema siempre permanecerá en un estado seguro, asi que antes de asignarle cualquier recurso, debo verificarlo.

Ver ejemplo página 218 del Silberschatz.

20.1.5.2 Algoritmo de grafo de asignación de recursos

Cuando tenemos un solo ejemplar de cada tipo de recursos, podemos usar una variante del grafo de asignación de recursos.

Además de la arista de solicitud y la de asignación, tenemos la de reserva. Va dirigida de P a R e indica que P podría solicitar R en un instante futuro. Se representa con una línea discontinua.

Los recursos deben reservarse a priori en el sistema, o sea que antes de iniciar la ejecución, ya deben aparecer todas las aristas de reserva.

La solicitud de un recurso solo puede ser satisfecha si la conversión de una arista discontinua en continua (asignación de recurso) no da lugar a ciclos.

20.1.5.3 Algoritmo del banquero

El grafo de asignación, no puede aplicarse a un sistema de asignación de recursos con múltiples ejemplares de cada recurso.

Cuando un proceso entra al sistema, debe declarar el número máximo de ejemplares de cada tipo de recursos que podría necesitar, el cual no puede exceder el total de recursos del sistema.

Cuando el usuario solicita un conjunto de recursos, el sistema debe determinar si la asignación no lo deja en un estado inseguro.

Hay que mantener varias estructuras de datos para implementar este algoritmo. Ellas son:

- **Disponible:** Un vector de longitud m indica el número de recursos disponibles de cada tipo. Disponible [j] = k, hay k ejemplares disponibles del recurso R_j.
- **Máx.:** Una matriz n x m define la demanda máxima de cada proceso. Máx. [i, j] = k, el proceso i puede solicitar como máximo k unidades del recurso j.
- **Asignación:** Una matriz n x m define el número de recursos que se han asignado actualmente a cada proceso. Asignación [i, j] = k. El proceso i tiene asignados actualmente k ejemplares del recurso j.
- **Necesidad:** Una matriz n x m indica los recursos que todavía le hacen falta a cada proceso.

Las estructuras de datos anteriores varían en el tiempo, tanto en tamaño como en valor.

20.1.5.4 Algoritmo de seguridad

1. Sean *trabajo* y *fin*, vectores con longitud m y n respectivamente. Inicializo:
   a. Trabajo:= Disponible
   b. Fin[i]:= false

2. Buscar una i tal que:
   a. Fin[i]= false
   b. $Necesidad_i \leq Trabajo$
   c. Si no existe i, continuar con el paso 4

3. Trabajo:= Trabajo + $Asignación_i$
   Fin[i] := true
   Ir al paso 2

4. Si Fin[i] = true para toda i, el sistema esta en estado seguro.

20.1.5.5 Algoritmo de solicitud de recursos

Sea $Solicitud_i$ el vector de solicitudes del proceso P_i. Si $Solicitud_i$ [j] = k, el proceso P_i quiere k ejemplares del tipo de recursos R_j. Para otorgarlos se siguen los siguientes pasos:

5. Si $Solicitud_i \leq Necesidad_i$, ir al paso 2, sino indicar error ya que el proceso excedió su reserva máxima.
6. Si $Solicitud_i \leq Disponible$, ir al paso 3, sino P_i debe esperar a que los recursos estén disponibles.

7. El sistema simula haber asignado los recursos al proceso P_i de la forma:
   Disponible:= *Disponible* – $Solicitud_i$;
   $Asignación_i$:= $Asignación_i$ + $Solicitud_i$;
   $Necesidad_i$:= $Necesidad_i$ – $Solicitud_i$;

Si el estado resultante es seguro, la transacción se lleva a cabo, asignando los recursos a P_i, sino P_i tendrá que esperar, y se restaura el antiguo estado de solicitud de recursos.

Ver ejemplo 7.5.3.3, página 222 del Silberschatz.

20.1.6 Detección de bloqueos mutuos

Si el sistema no previene o evita bloqueos mutuos, se puede encontrar con una situación de bloqueo. Para ello necesita:

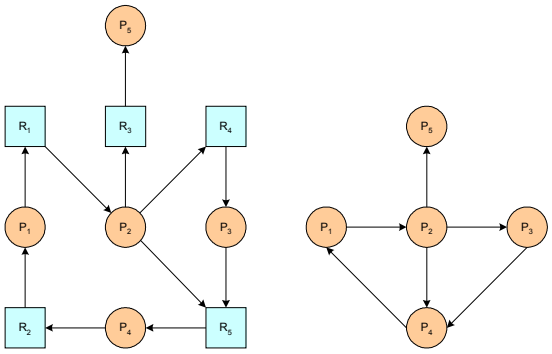
- Un algoritmo que examine el estado del sistema y determine si hay bloqueo
- Un algoritmo para recuperarse del bloqueo

Este esquema de detección y recuperación requiere un gasto extra que incluye no solo los costos en tiempo para mantener la información sino también para recuperarse después del bloqueo.

20.1.6.1 Un solo ejemplar de cada tipo de recursos

Se utiliza el *grafo de espera*, que se obtiene eliminando del grafo de asignación de recursos, los nodo del tipo de recursos, y uniendo las aristas apropiadas.

Una arista de P_i a P_j implica que el proceso i, espera a que j libere un recurso que i necesita.



Grafo de asignación de recursos — Grafo de espera correspondiente

Existe un bloqueo mutuo ⇔ en el grafo hay un ciclo. El sistema necesita mantener la información del grafo, y periódicamente invocar el algoritmo de detección de ciclos.

20.1.6.2 Varios ejemplares de cada tipo de recursos

Para este caso se utilizan estructuras similares a las del algoritmo del banquero.

- **Disponible:** Un vector de longitud m indica el número de recursos disponibles de cada tipo.
- **Asignación:** Una matriz n x m define el número de recursos de cada tipo ya asignados a cada proceso.
- **Solicitud:** Una matriz n x m indica la solicitud actual de cada proceso. Solicitud [i, j] = k → P_i solicita k ejemplares de P_j.

1. Sean *trabajo* y *fin* vectores con longitud m y n respectivamente. Inicializo:
   a. Trabajo:= Disponible

   b. Fin[i]:= false si Asignación[i] <> 0, sino Fin[i]:= true

2. Busco i tal que
   a. Fin[i] = false
   b. $Solicitud_i \leq Trabajo$
   c. Si no existe i, salto a 4

3. $Trabajo := Trabajo + Asignacion_i$
   Fin[i]:= true
   Ir al paso 2

4. Si Fin[i] = false para alguna i → El sistema está en bloqueo mutuo para ese i.

   Ver ejemplo página 226 del Silberschatz.

20.1.6.3 Uso del algoritmo de detección

Si los bloqueos mutuos son frecuentes el algoritmo de detección deberá invocarse muy seguido ya que los recursos asignados a los procesos bloqueados están ociosos hasta romper ese bloqueo, además el número de procesos bloqueados, puede crecer.

Los bloqueos mutuos solo pueden aparecer cuando algún proceso pide un recurso y no se le puede asignar inmediatamente. Se podría invocar el algoritmo, cada vez que esto sucede, además podemos identificar inmediatamente el proceso causante del bloqueo. Pero invocarlo tan frecuentemente, puede causar demasiado gasto extra de tiempo, por lo tanto se lo puede invocar luego de un tiempo determinado (Ej: 1 hora) o cuando el uso del CPU baja del 40%. En este caso pueden existir muchos ciclos en el grafo, y se dificulta identificar los procesos causantes del bloqueo.

20.1.7 Recuperación después del bloqueo mutuo

Cuando un algoritmo detecta un bloqueo puede:

- Informar al operador para que lo solucione manualmente
- Dejar que el SO se recupere automáticamente

Las opciones para romper el bloqueo son terminar anormalmente los procesos, o expropiarle recursos.

20.1.7.1 Terminación de procesos

Hay dos formas:

- **Abortar todos los procesos bloqueados:** El costo es elevado puesto que alguno de los procesos puede haber estado trabajando por largo tiempo.

- **Abortar de a un proceso hasta desbloquearlos:** Incluye un gasto adicional, puesto que luego de eliminar cada proceso hay que llamar nuevamente al algoritmo de detección.

La terminación forzada de procesos no es fácil, puesto que podría dejar incoherencia de datos.

Para saber que procesos del bloqueo se pueden abortar, se debería usar un algoritmo similar al de la planificación de CPU. Se deben abortar los procesos causando el más bajo costo posible.

Los factores para selección de procesos son:

- Prioridad del proceso
- Tiempo que ha trabajado, y cuanto más debe trabajar
- Recursos usados y de que tipo (si los recursos se pueden expropiar fácilmente)
- Recursos adicionales que necesita para terminar su tarea
- Cantidad de procesos para abortar
- Si los procesos son interactivos o por lotes

20.1.7.2 Expropiación de recursos

Expropiamos recursos de unos procesos para dárselos a otros, hasta romper el bloqueo. Hay que considerar tres aspectos:

- **Selección de la víctima:** Debemos determinar el orden de expropiación para minimizar el costo.
- **Retroceso:** Si expropiamos el recurso, el proceso que lo estaba usando, no puede continuar con su ejecución normal, por lo tanto, el proceso debe retroceder a un estado seguro, y reiniciarlo. La forma más fácil es abortar el proceso y comenzar de nuevo, pero es más conveniente regresar el proceso a un estado seguro, aunque se debe tener guardada información para ello.
- **Inanición:** Podría suceder que siempre se escoja el mismo proceso como víctima, por lo tanto hay que asegurarse que solo se lo va a expropiar un número finito (pequeño) de veces.

20.1.8 Estrategia combinada para el manejo de bloqueos mutuos

Ninguna de las estrategias anteriores, por si sola es apropiada. Una posibilidad es combinar los tres enfoques básicos y usar el óptimo para cada clase de recursos. Para ello, se pueden ordenar jerárquicamente los recursos, usando dentro de cada clase, la técnica más adecuada, evitando así los bloqueos mutuos.

Clasificación:

- **Recursos internos:** Recursos utilizados por el sistema, como un bloque de control de procesos.
- **Memoria central:** Memoria utilizada por trabajos de usuario.
- **Recursos de trabajos:** Dispositivos asignables y archivos.
- **Espacio intercambiable:** Espacio para cada trabajo de usuario en el almacenamiento auxiliar.

Una solución mixta ordena las clases de la siguiente forma, utilizando las siguientes estrategias:

- **Recursos internos:** Se puede usar prevención mediante ordenamiento de recursos, ya que no es necesario escoger entre solicitudes pendientes en el momento de la ejecución.
- **Memoria central:** Se puede usar prevención mediante expropiación ya que los trabajos pueden intercambiarse a disco, y la memoria central puede expropiarse.
- **Recursos de trabajos:** Se puede usar evitación, ya que la información requerida acerca de las necesidades de recursos, se puede obtener de las tarjetas de control de trabajos.
- **Espacio intercambiable:** Se puede usar preasignación, pues casi siempre se conocen las necesidades de almacenamiento máximas.

21 Sistema de archivos:

Ejercicios parciales y examenes:

21.1.1 segundo parcial 2002:

Problema 1 (18 puntos)

Se desea implementar el sistema de archivos de un sistema operativo, de tal manera que los siguientes requerimientos sean soportados.

- Manejo jerárquico de directorios, con cualquier profundidad.
- Un máximo de MAX_ARCHIVOS_DIR entradas por directorios.
- Manejo de archivos, donde estos deben soportar:

_ nombres con un largo máximo de 8 caracteres.
_ extensiones con un largo máximo de 3 caracteres.
_ información descriptiva de cada archivo (hasta 10 caracteres).
_ dirección de comienzo en la FAT del almacenamiento del archivo.

- Manejo de subdirectorios:

_ nombres con un largo máximo de 8 caracteres.

Se guardan los datos de los archivos en una estructura tipo FAT, la cual tiene una cantidad limitada de sectores en disco (MAX_SECTORES_DISCO). Cada elemento de la FAT corresponde a un sector del disco. Cuando ese sector no está utilizado el elemento correspondiente de la FAT contiene cero. En caso de que el sector este asignado a un archivo pero no sea el último del mismo, contiene la dirección del siguiente sector asignado a dicho archivo. Cuando es el último sector asignado a un archivo, contiene FFFF. Un sector tiene TAM_SECTOR bytes.
Se pide:
a) Indique las estructuras necesarias para satisfacer estos requerimientos (incluyendo los directorios y la FAT).
b) Indique como será cargada esta estructura al iniciar el sistema y donde será almacenada en el disco esta misma. Indique también si
existe alguna restricción en el lugar donde se almacena su estructura.
c) Dado un archivo, indicado por su camino absoluto (desde el directorio raíz), escribir un algoritmo que encuentre dicho un archivo, y
que luego lo recorra, devolviendo la cantidad de bytes que ocupa
d) Indique tres posibles consecuencias de una caída de la máquina (por ejemplo, por un corte de energía) en cuanto a la integridad de
las estructuras definidas.
Aclaraciones:
-Se dispone de la función auxiliar parseCamino, la cual dada una ruta absoluta para un archivo, retorna un array de strings con los
componentes del camino junto con la cantidad de componentes en dicho camino
parseCamino(char[], char[][], integer)
Por ejemplo, la invocación: parseCamino("/dir1/dir12/dir123/arch.ext", ruta, cntcmp);
Deja ruta cargado con la siguiente información: ruta = { "/", "dir1", "dir12", "dir123", "arch.ext" } y cntcmp con
el valor 5. Si existe alguna condición de error, ruta vuelve NULL y cntcmp en -1

-Los archivos indicados por un camino absoluto siempre empiezan con "/", y terminan con el nombre del archivo
-Se podrá disponer de las funciones
writeDirectoryToSector(dir: *** ; sector: integer);
readDirectoryFromSector(sector: integer): ***;
Las cuales leen y graban un directorio completo a un sector del disco. Se asume que un directorio entero ocupa un sector. Asimismo la
escritura de un directorio, no implica la escritura de los subdirectorios en este presente.
*** representa la estructura que se defina para los directorios
-Se dispone de la función:
directoryIsLoaded(char[] dir_name): ***;
La cual indica si un directorio a sido o no cargado desde el disco. En caso afirmativo devuelve el directorio cargado, sino devuelve NULL.
El propio sistema operativo es el encargado de mantener en sus estructuras de memoria los directorios cargados.
*** representa la estructura que se defina para los directorios
-Se dispone de las funciones:
writeSector(data: byte[TAM_SECTOR], sector: integer);
readSector(sector: integer, data: byte[TAM_SECTOR]);
==========

Solución:

```
a) Las siguientes estructuras cumplen con los requisitos
pedidos.
Representamos las entradas de directorios, donde cada una puede
representar un archivo o un subdirectorio. En cualquier caso
tenemos una
marca de booleana de usado.
Para los archivos guardamos nombre, extension, informacion
descriptiva y el
primer sector de la FAT donde se encuentra la informacion del
mismo. Luego se
encadena en la FAT la informacion que el archivo contiene.
Para los directorios lo que tenemos es una entrada para el
nombre y el sector
en el disco donde se almacena esta estructura.
type direntry = record
      used : boolean;
      case type:(file, directory) of
      file:
            file_name: array [1..8] of char;
            file_ext: array [1..3] of char;
            info: array [1..100] of char;
            first_file_sector: integer;
      directory:
            dir_name: array [1..8] of char;
            dir_sector: integer;
            end_case;
end;
type dir = array[1..MAX_ARCHIVOS_DIR] of direntry;
type fat = array[1..MAX_SECTORES_DISCO] of integer;

b) Al iniciar el sistema deberan levantarse la estructura para
la FAT y para el directorio raiz, ya que estos son los puntos
iniciales desde donde se navega el sistema de archivos. Para
esto el directorio raiz sera almacenado en el sector 0 del
```

disco, indicandose en la FAT este sector siempre como ocupado. La FAT sera almacenada en los sectores siguientes del disco, ocupando tantos sectores como corresponda al tamaño de la estructura. La FAT debera ser respaldada a disco antes de finalizar el
trabajo con el sistema para evitar inconsistencias en el sistema de archivos. Asimismo, al inciar el sistema, debera dejarse disponible en forma global la FAT y el directorio raiz, DIRECTORIO_RAIZ.

c) Nos pasan un archivo especificado por su camino absoluto. A partir de este debemos recorrer desde la raiz del sistema de archivos

```
function recorre_Y_tamanio(char[] rutain): integer;
var
     char ruta[][];
     integer cntcmp, i, j, tamanio;
     dir diractual;
     direntry de;
     char dirNameActual[][];
begin
// Empezamos en el directorio raiz
diractual = DIRECTORIO_RAIZ;
dirNameActual = “/”;
// Obtenemos los componentes del camino
parseCamino(rutain, ruta, cntcmp);
// No existe la ruta
if (cntcmp = -1)
     return –1;
end if
// Con la informacion obtenida, cambiamos de
// directorio hasta llegar al directorio del
// archivo. Recordamos que “/” y “nomarch”
// siempre vienen en el resultado
for i = 2 to cntcmp-1 begin
     j = localizaEntrada(diractual, ruta[i], “D”);
     // No existe la entrada o me la borraron
     if (j = -1)
          return –1;
     end if
     // Encontramos el directorio, cambiamos y seguimos
     de = diractual[j];
     dirNameActual= dirNameActual+de.dirname;
     if not (de.used and de.type = directory)
          return –1;
     else
          // Si no esta cargado, lo cargamos
          diractual = directoryIsLoaded(de.dir_name)
          if (diractual = NULL)
               diractual = readDirectoryFromSector
          (dirNameActual);
          end if
     end if
end for
// Despues de recorrer toda la cadena, llegamos al archivo
// Procedemos a procesarlo
j = localizaEntrada(diractual, ruta[cntcmp], “A”);
```

```
// No existe la entrada o me la borraron
if (j = -1)
     return –1;
end if
de = diractual[j];
if not (de.used and de.type = file)
     return –1;
else
     tamanio = recorre_y_calcula(de.file_first_sector);
end if
end;
// Funciones auxiliares.
// Localiza una entrada de directorio, tanto archivo como
// subdirectorio
function localizaEntrada(dir da, char[] name, char typ): integer
var
     integer pos, i;
     boolean found;
begin
pos = -1;
i = 1;
found = false;
while ((not found) and i <= MAX_ARCHIVOS_DIR) do begin
     if (da[i].used and da[i].type = file and typ = “A”)
          char[] nom = da[i].file_name + “.” + da[i].file_ext;
               if (nom = name)
                    pos = i;
                    found = true;
               end if
     end if
     if (da[i].used and da[i].type = directory and typ = “D”)
          char[] nom = da[i].dir_name;
          if (nom = name)
               pos = i;
               found = true;
          end if
     end if
     i = i + 1
end while
return pos;
end;

// Se recorre un archivo pasando por todos sus sectores y
// contando el total de bytes

function recorre_Y_calcula(integer start_sector): integer
begin
int sec = start_sector;
int cnt = 1;
while FAT[sec] <> FFFF do
     cnt = cnt + 1;
     sec = FAT[sec];
end while
return cnt * TAM_SECTOR;
end
```

d) Consecuencias en cuanto a las estructuras definidas ante una caida del sistema:

```
1.- Corrupcion en los archivos del disco. Una cadena de sectores
en la FAT puede quedar sin el marcador de fin FFFF, con lo cual
no se sabria donde termina el archivo.
2.- Dos cadenas de archivos pueden unirse (tambien por
corrupcion) en un mismo archivo.
3.- Pueden existir cadenas ocupadas, que no pertenezcan a nigun
archivo
4.- Los directorios pueden no ser almacenados en disco ante una
caida
================================================================
```

21.1.2 Práctico 8 - Ejercicio 1

Un sistema operativo maneja sus archivos en disco utilizando las siguientes estructuras:

```
type dir = array [1..max_files_on_disk] of record
      used : boolean
            {entrada usada o no}
      file_name : array [1..8] of char;
      file_ext : array [1..3] of char;
      informacion : array [1..100] of char;
      comienzo : integer {primer sector asignado al archivo}
end
type fat = array [1..sectors_in_disk] of integer
```

Cada elemento de la `fat` corresponde a un sector del disco. Cuando ese sector no está utilizado el elemento correspondiente de la `fat` contiene cero. En caso de que el sector este asignado a un archivo pero no sea el último del mismo, contiene la dirección del siguiente sector asignado a dicho archivo. Cuando es el último sector asignado a un archivo, contiene FFFF.

a) Escribir un algoritmo que encuentre en el directorio un archivo, y lo recorra (es decir, lea en orden todos sus sectores).

b) Escribir un algoritmo que cree e inicialice con ceros un archivo de *n* bytes (un sector tiene 512 bytes).

c) Discutir las posibles consecuencias de una caída de la máquina (por ejemplo, por corte de energía) en cuanto a la integridad de las estructuras definidas.

d) Escribir un algoritmo que chequee la consistencia de las estructuras de un disco y que, de ser posible, corrija sus anomalías.

e) Escribir un algoritmo que reorganice el disco de manera que todo archivo quede almacenado en sectores contiguos.

Explicar la conveniencia de esto último.

```
================================================================
var laFat: Fat; elDir: Dir;
a)
procedure encontrarRecorrer(elDir: Dir; archivo: File);
encontre: bolean; indArch, sectorAct: integer;
begin
{busco el archivo en el directorio}
indArch := encontrar(elDir,archivo);
{si lo encontre lo recorro}
if indArch>0 then
 begin
      sectorAct := elDir[indArch].comienzo;
      while (laFat[sectorAct]<>0xFFFF) do
       begin
            leerDeDisco(sectorAct);
            sectAct := laFat[sectorAct];
```

```
      end
 end
end

b)
function crearEnCero(archivo: File; tam: Integer):Integer;
iterFAT, indDir, primero: integer;
begin
  {precondición: no existe el archivo}
  indDir := encontrarDirLibre();
  if (indDir>=0) then
  begin
       primero := armarCadena(tam);
       if primero>=0 then
       begin
       elDir[indDir].used := true;
       elDir[indDir].file_name := archivo.nombre;
       elDir[indDir].file_ext := archivo.extension;
       elDir[indDir].comienzo := primero;
       crearEnCero := 0;
       end
       else crearEnCero := -1
  end
  else crearEnCero := -1
end
c)
```

- Ciclos en una de las listas encadenas.
- Que dos archivos compartan su cola.
- Que una la lista contenga sectores defectuosos.
- Entradas en la fat no válidas (valores reservados, valores mayores que el tope, entradas que deberían estar libres pero aparecen marcadas).
- Un archivo no disponga de entrada fin de archivo en la fat.

d) Implementar un algoritmo que corrija las situaciones encontradas en la parte c). Ejemplos: fschk, chkdsk, scandisk, etc.

e) Ejemplos: defrag.

Funciones auxiliares

```
function encontrar(archivo: File): interger;
encontre: bolean; iterArch: integer;
begin
  {busco el archivo en el directorio}
  encontre:= false; iterArch := 1;
  while not encontre and iterArch<=max_files_on_disk do
  begin
    encontre := elDir[iterArch].used and
    (elDir[iterArch].file_name=archivo.nombre) and
    (elDir[iterArch].file_ext=archivo.extension);
    iterArch := iterArch + 1;
  end
  if encontre then
     encontrar := iterArch – 1;
  else
     encontrar := -1;
end
```

```
function armarCadena(tam: integer): integer;
iterFAT, sectAnt, primero: integer; libre: bolean;
begin
  primero := encontrarFATLibre(1);
  error := primero;
  tam := tam – 512;
  sectAnt := primero;
  while tam>0 and error>=0 do
  begin
    iterFat := encontrarFATLibre(sectAnt+1);
    if iterFat>0 then
    begin
          tam := tam – 512;
          FAT[sectAnt]:=iterFAT;
          SectAnt := iterFAT;
    end
    else error := -1;
  end
  if error>=0 then
     begin
          FAT[sectAnt] := 0xFFFF;
          armarCadena := primero;
     end
  else
  begin
   while primero<>sectAnt do
    begin
     sectTmp := primero;
     primero := FAT [primero];
     FAT[sectTmp] := 0;
    end;
   FAT[sectAnt] := 0;
   armarCadena := -1;
  end
end

function encontrarFATLibre(iter: integer): interger;
encontre: bolean;
begin
{busco sector libre en la FAT}
encontre:= false;
while not encontre and iter<=sectors_on_disk do
begin
     encontre := (fat[iter]=0);
     iter := iter + 1;
end
if encontre then
     encontrarFATLibre:= iter – 1;
else
     encontrarFATLibre:= -1;
end
function encontrarDirLibre(): interger;
encontre: bolean; iterArch: integer;
begin
     {busco entrada libre en el directorio}
     encontre:= false; iterArch := 0;
          while not encontre and iterArch<=max_files_on_disk do
```

```
            begin
                  encontre := not elDir[iterArch].used;
                  iterArch := iterArch + 1;
            end
      if encontre then
            encontrarDirLibre:= iterArch – 1;
      else
            encontrarDirLibre:= -1;
      end
=================================================================
```

Directorios de archivos:

[sta]

Contenido

Asociado con cualquier sistema de gestión de archivos o cualquier colección de archivos suele haber un directorio de archivos. El directorio contiene información sobre los archivos, incluyendo atributos, ubicación y propietario. Gran parle de esta información, especialmente la relativa al almacenamiento, la gestiona el sistema operativo. El directorio es propiamente un archivo, poseído por el sistema operativo y accesible a través de diversas rutinas de gestión de archivos. Aunque parte de la información de los directorios está disponible para los usuarios y aplicaciones, en general, la información se proporciona indirectamente, a través de rutinas del sistema. De este modo los usuarios no pueden acceder directamente al directorio, incluso en modo de sólo lectura.

TABLA 11.2 Elementos de información de un directorio de archivos

	Información Básica
Nombre del archivo	Nombre elegido por el creador (usuario o programa). Debe ser único en un directorio específico.
Tipo de Archivo	Por ejemplo: texto, binario, módulo de carga, etc.
Organización del Archivo	Para sistemas que soportan varias organizaciones.
	Información de Direccionamiento
Volumen	Indica el dispositivo donde se almacena el archivo.
Dirección de Comienzo	Dirección física de inicio en memoria secundaria (cilindro, pista y número de bloque en disco).
Tamaño Usado	Tamaño actual del archivo en bytes, palabras o bloques.
Tamaño Asignado	Tamaño máximo del archivo.
	Información de Control de Acceso
Propietario	Usuario con control sobre el archivo. El propietario puede otorgar o denegar acceso a otros usuarios y cambiar estos privilegios.
Información de Acceso	Una versión simple de este elemento incluye el nombre del usuario y la contraseña para cada usuario autorizado.
Acciones Permitidas	Controla la lectura, escritura, ejecución y transmisión por una red.
	Información de Uso
Fecha de Creación	Cuándo se añadió el archivo al directorio
Identidad del Creador	Normalmente, pero no siempre, el propietario.
Fecha de Última Lectura	Fecha de la última vez que se leyó un registro.
Identidad del Último Lector	Usuario que hizo la lectura.
Fecha de Última Modificación	Fecha de la última actualización, inserción o borrado.
Identidad del Último Modificador	Usuario que hizo la modificación
Fecha de la Última Copia de Seguridad	Fecha de la última vez que el archivo fue copiado en otro medio de almacenamiento.
Utilización Actual	Información sobre la actividad actual sobre el archivo, tal como el (los) proceso(s) que tienen abierto el archivo, si está bloqueado por un proceso y si el archivo ha sido actualizado en memoria principal, pero aún no en disco.

En la tabla 11.2 se propone la información que se almacena normalmente en el directorio para

cada archivo del sistema. Desde el punto de vista del usuario, el directorio ofrece una traducción
entre los nombres de archivo conocidos para usuarios y aplicaciones y los archivos, propiamente dicho. Por tanto, cada entrada incluirá el nombre del archivo. Casi todos los sistemas trabajan con clases diferentes de archivos y diferentes organizaciones de archivos, por lo que también se incluye esta información. Un tipo de información importante sobre cada archivo es aquella relativa a su almacenamiento, incluyendo su ubicación y tamaño. En los sistemas compartidos, también es importante ofrecer información para controlar el acceso al archivo. Normalmente, un usuario será el propietario del archivo y podrá otorgar ciertos privilegios de acceso a otros usuarios. Finalmente, se necesita información sobre su uso para gestionar la utilización actual del archivo y guardar un histórico.

A fin de organizar los archivos, los sistemas de archivos casi siempre tienen **directorios** que en muchos sistemas, son también archivos. En esta sección hablaremos de los directorios, su organización, sus propiedades y las operaciones que pueden efectuarse con ellos.

[tan]

21.1.3 Sistemas de directorio jerárquicos

Un directorio normalmente contiene varias entradas, una por archivo. Una posibilidad se muestra en la Fig. 5-5(a), donde cada entrada contiene el nombre del archivo, los atributos del archivo, y la dirección de disco donde están almacenados los datos. Otra posibilidad se muestra en la Fig. 5-5(b). Aquí una entrada de directorio contiene el nombre del archivo y un apuntador a otra estructura de datos en la que pueden encontrarse los atributos y las direcciones en disco. Ambos sistemas son de uso generalizado.

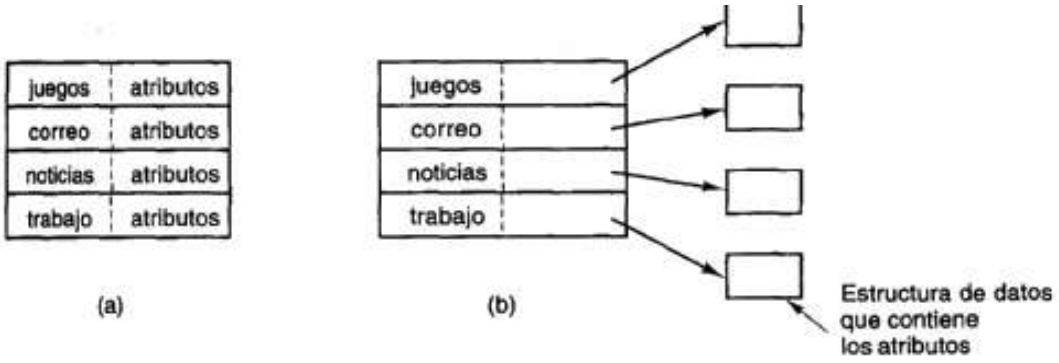


Figura 5-5. (a) Atributos en la entrada de directorio. (b) Atributos en otro lugar.

Cuando se abre un archivo, el sistema operativo examina su directorio hasta encontrar el nombre del archivo por abrir, y luego extrae los atributos y las direcciones en disco, ya sea directamente de la entrada de directorio o de la estructura de datos a la que ésta apunta, y los coloca en una tabla en la memoria principal. Todas las referencias subsecuentes al archivo utilizan la información que está en la memoria principal.
El número de direcciones varía de un sistema a otro. El diseño más sencillo es aquel en el que el
sistema mantiene un solo directorio que contiene todos los archivos de todos los usuarios, como se ilustra en la Fig. 5-6(a). Si hay muchos usuarios, y éstos escogen los mismos nombres de archivo (p. ej., correo y juegos), los conflictos y la confusión resultante pronto harán

inmanejable el sistema. Este modelo se utilizó en los primeros sistemas operativos de microcomputadoras, pero ya casi nunca se encuentra.
Una mejora de la idea de tener un solo directorio para todos los archivos del sistema es tener un directorio por usuario [véase la Fig. 5-6(b)]. Este diseño elimina los conflictos de nombres entre usuarios pero no es satisfactorio para quienes tienen un gran número de archivos. Es muy común que los usuarios quieran agrupar sus archivos atendiendo a patrones lógicos. Un profesor, por ejemplo, podría tener una colección de archivos que juntos constituyan un libro que está escribiendo para un curso, una segunda colección de archivos que contenga programas que los estudiantes de otro curso hayan presentado, un tercer grupo de archivos que contenga el código de un sistema avanzado de escritura de compiladores que él esté construyendo, un cuarto grupo de archivos que contenga propuestas para obtener subvenciones, así como otros archivos para correo electrónico, minutas de reuniones, artículos que esté escribiendo, juegos, etc. Se requiere algún método para agrupar estos archivos en formas flexibles elegidas por el usuario.

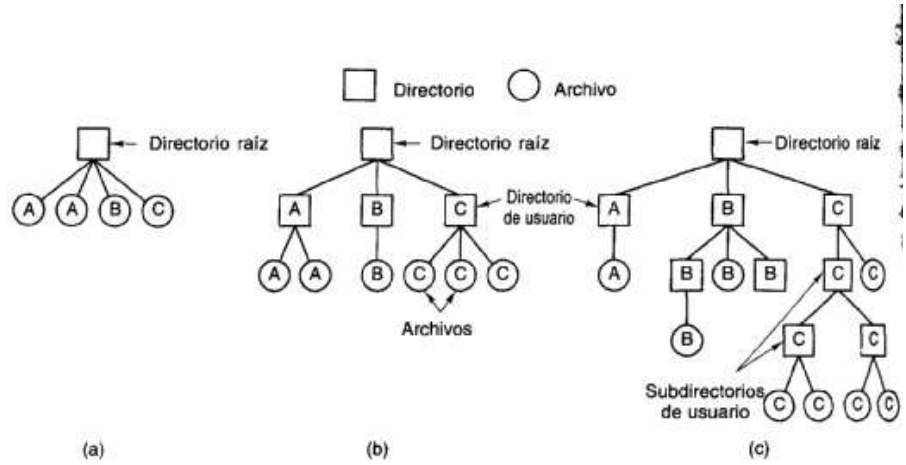


Figura 5-6. Tres diseños de sistema de archivos. (a) Un solo directorio compartido por todos los usuarios. (b) Un directorio por usuario. (c) Árbol arbitrario por usuario. Las letras indican el propietario del directorio o archivo.

Lo que hace falta es una jerarquía general (es decir, un árbol de directorios). Con este enfoque, cada usuario puede tener tantos directorios como necesite para poder agrupar sus archivo) según patrones naturales. Este enfoque se muestra en la Pig. 5-6(c). Aquí, los directorios A, ¿?| C, contenidos en el directorio raíz, pertenecen cada uno a un usuario distinto. Dos de estos usuarios han creado subdirectorios para proyectos en los que están trabajando.

21.1.4 Implementación de sistemas de archivos:

Ya es momento de pasar de la perspectiva del usuario del sistema de archivos a la perspectiva del implementador. A los usuarios les interesa la forma de nombrar los archivos, las operaciones que pueden efectuarse con ellos, el aspecto que tiene el árbol de directorios y cuestiones de interfaz por el estilo. A los implementadores les interesa cómo están almacenados los archivos y directorios, cómo se administra el espacio en disco y cómo puede hacerse que todo funcione de forma eficiente y confiable. En las secciones siguientes examinaremos varias de estas áreas para conocer los problemas y las concesiones.

21.1.4.1 Implementación de archivos

Tal vez el aspecto más importante de la implementación del almacenamiento en archivos sea poder relacionar bloques de disco con archivos. Se emplean diversos métodos en los diferentes sistemas operativos. En esta sección examinaremos algunos de ellos.

21.1.4.2 Asignación contigua

El esquema de asignación más sencillo es almacenar cada archivo como un bloque contiguo de datos en el disco. Así, en un disco con bloques de 1K, a un archivo de 50K se le asignarían 50 bloques consecutivos. Este esquema tiene dos ventajas importantes. Primera, la implementación es sencilla porque para saber dónde están los bloques de un archivo basta con recordar un número, la dirección en disco del primer bloque. Segunda, el rendimiento es excelente porque es posible leer todo el archivo del disco en una sola operación.
Desafortunadamente, la asignación contigua tiene también dos ventajas igualmente importantes.
Primera, no es factible si no se conoce el tamaño máximo del archivó en el momento en que se crea el archivo. Sin esta información, el sistema operativo no sabrá cuánto espacio en disco debe reservar. Sin embargo, en los sistemas en los que los archivos deben escribirse de un solo golpe, el método puede usarse con gran provecho.
La segunda desventaja es la fragmentación del disco que resulta de esta política de asignación. Se desperdicia espacio que de otra forma podría haberse aprovechado. La compactación del disco suele tener un costo prohibitivo, aunque tal vez podría efectuarse de noche cuando el sistema estaría ocioso.

21.1.4.3 Asignación por lista enlazada

El segundo método para almacenar archivos es guardar cada uno como una lista enlazada de bloques de disco, como se muestra en la Fig. 5-8. La primera palabra de cada bloque se emplea como apuntador al siguiente. El resto del bloque se destina a datos.

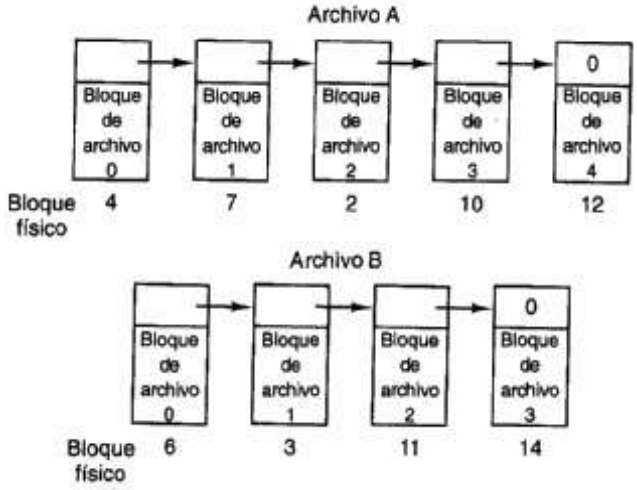


Figura 5-8. Almacenamiento de un archivo como lista enlazada de bloques de disco.

A diferencia de la asignación contigua, con este método es posible utilizar todos los bloques. No
se pierde espacio por fragmentación del disco (excepto por fragmentación interna en el último bloque). Además, basta con que en la entrada de directorio se almacene la dirección en disco del primer bloque. El resto del archivo puede encontrarse siguiendo los enlaces.
Por otro lado, aunque la lectura secuencial de un archivo es sencilla, el acceso aleatorio es extremadamente lento. Además, la cantidad de almacenamiento de datos en un bloque ya no es una potencia de dos porque el apuntador ocupa unos cuantos bytes. Si bien tener un tamaño peculiar no es fatal, resulta menos eficiente porque muchos programas leen y escriben en bloques cuyo tamaño es una potencia de dos.

21.1.4.4 Asignación por lista enlazada empleando un índice

Las dos desventajas de la asignación por lista enlazada pueden eliminarse si se toma la palabra de apuntador de cada bloque y se le coloca en una tabla o índice en la memoria. La Fig. 5-9 muestra el aspecto que la tabla tendría para el ejemplo de la Fig. 5-8. En ambas figuras, tenemos dos archivos. El archivo A usa los bloques de disco 4,7,2,10 y 12, en ese orden, y el archivo B usa los bloques de disco 6,3, 11 y 14, en ese orden. Si usamos la tabla de la Fig. 5-9, podemos comenzar en el bloque 4 y seguir la cadena hasta el final. Lo mismo puede hacerse comenzando en el bloque 6.

Figura 5-9. Asignación por lista enlazada empleando una tabla en la memoria principal.

Si se emplea esta organización, todo el bloque está disponible para datos. Además, el acceso directo es mucho más fácil. Aunque todavía hay que seguir la cadena para encontrar una distancia dada dentro de un archivo, la cadena está por completo en la memoria, y puede seguirse sin tener que consultar el disco. Al igual que con el método anterior, basta con guardar un solo entero (el número del bloque inicial) en la entrada de directorio para poder localizar todos los bloques, por más grande que sea el archivo. MS-DOS emplea este método para la asignación en disco.

La desventaja primordial de este método es que toda la tabla debe estar en la memoria todo el tiempo para que funcione. En el caso de un disco grande con, digamos, 500 000 bloques de 1K (500M), la tabla tendrá 500 000 entradas, cada una de las cuales tendrá que tener un mínimo de 3 bytes. Si se desea acelerar las búsquedas, se necesitarán 4 bytes. Así, la tabla ocupará de 1.5 a 2 megabytes todo el tiempo, dependiendo de si el sistema se optimiza en cuanto al espacio o en cuanto al tiempo. Aunque MS-DOS emplea este mecanismo, evita manejar tablas muy grandes empleando bloques grandes (de hasta 32K) en los discos de gran tamaño.

21.1.4.5 Nodos-i

Nuestro último método para saber cuáles bloques pertenecen a cuál archivo consiste en asociar a cada archivo una pequeña tabla llamada **nodo-i** (**nodo-índice**), que lista los atributos y las direcciones en disco de los bloques del archivo, como se muestra en la Fig. 5-10

Las primeras pocas direcciones de disco se almacenan en el nodo-i mismo, así que en el caso de archivos pequeños toda la información está contenida en el nodo-i, que se trae del disco a la memoria principal cuando se abre el archivo. En el caso de archivos más grandes, una de las direcciones del nodo-i es la dirección de un bloque de disco llamado bloque de indirección sencilla. Este bloque contiene direcciones de disco adicionales. Si esto todavía no es suficiente, otra dirección del nodo-i, llamada bloque de indirección doble, contiene la dirección de un

bloque que contiene una lista de bloques de indirección sencilla. Cada uno de estos bloques de indirección sencilla apunta a unos cuantos cientos de bloques de datos. Si ni siquiera con esto basta, se puede usar también un **bloque de dirección triple**. UNIX emplea este esquema.

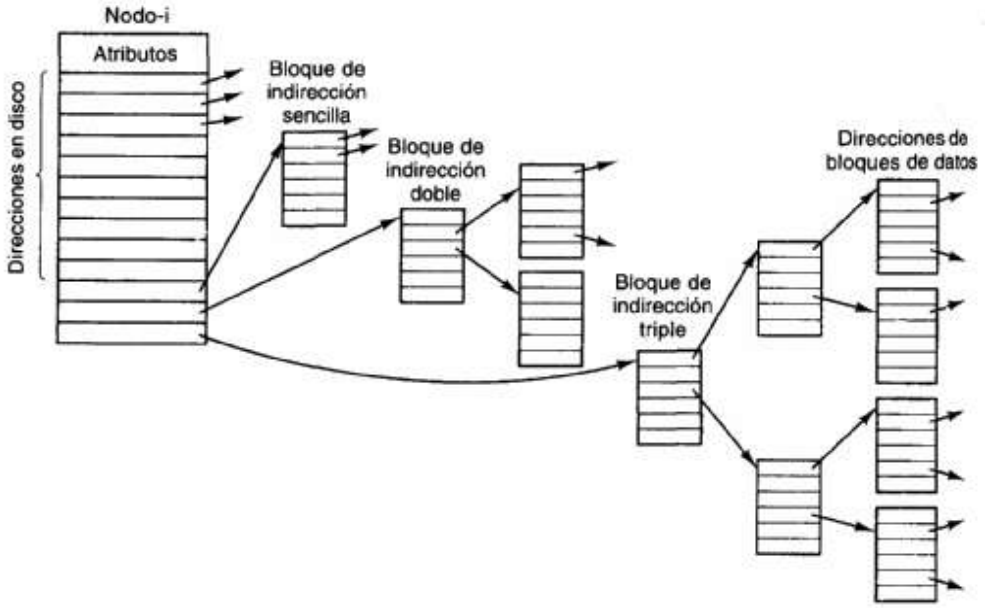


Figura 5-10. Un nodo-i.

21.1.5 Implementación de directorios

Antes de poder leer un archivo, hay que abrirlo. Cuando se abre un archivo, el sistema operativo usa el nombre de ruta proporcionado por el usuario para localizar la entrada de directorio. Esta entrada proporciona la información necesaria para encontrar los bloques de disco. Dependiendo del sistema, esta información puede ser la dirección en disco de todo el archivo (asignación contigua), el número del primer bloque (ambos esquemas de lista enlazada) o el número del nodo-i. En todos los casos, la función principal del sistema de directorios es transformar el nombre ASCII del archivo en la información necesaria para localizar los datos. Un problema muy relacionado con el anterior es dónde deben almacenarse los atributos. Una posibilidad obvia es almacenarlos directamente en la entrada de directorio. Muchos sistemas hacen precisamente esto. En el caso de sistemas que usan nodos-i, otra posibilidad es almacenar los atributos en el nodo-i, en lugar de en la entrada de directorio. Como veremos más adelante, este método tiene ciertas ventajas respecto a la colocación de los atributos en la entrada de directorio.

21.1.5.1 Directorios en CP/M

Iniciemos nuestro estudio de los directorios con un ejemplo especialmente sencillo, el de CP/M (Golden y Pechura, 1986), ilustrado en la Fig. 5-11. En este sistema, sólo hay un directorio, así que todo lo que el sistema de archivos tiene que hacer para consultar un nombre de archivo es buscarlo en el único directorio. Una vez que encuentra la entrada, también tiene los números de bloque en el disco, ya que están almacenados ahí mismo, en la entrada, lo mismo que todos los atributos. Si el archivo ocupa más bloques de disco de los que caben en una entrada, se asignan al archivo entradas de directorio adicionales.
Los campos de la Fig. 5-11 tienen los siguientes significados. El campo de código de usuario indica qué usuario es el propietario del archivo. Durante una búsqueda, sólo se examinan las entradas que pertenecen al usuario que está actualmente en sesión. Los siguientes dos campos

dan el nombre y la extensión del archivo. El campo alcance (extent) es necesario porque un archivo

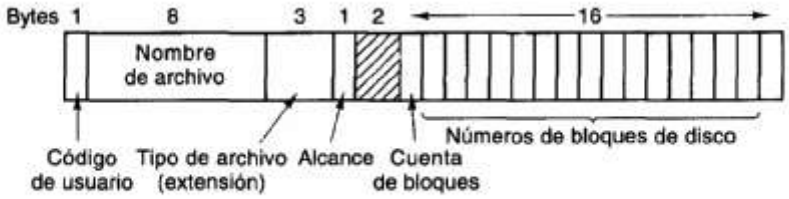


Figura 5-11. Entrada de directorio que contiene los números de bloque de disco para cada archivo.

Los campos de la Fig. 5-11 tienen los siguientes significados. El campo de código de usuario indica qué usuario es el propietario del archivo. Durante una búsqueda, sólo se examinan las entradas que pertenecen al usuario que está actualmente en sesión. Los siguientes dos campos dan el nombre y la extensión del archivo. El campo alcance (extent) es necesario porque un archivo con más de 16 bloques ocupa múltiples entradas de directorio. Este campo sirve para saber cuál entrada es primero, cuál segunda, etc. El campo de cuenta de bloques indica cuántas de las 16 posibles entradas de bloque se están usando. Los últimos 16 campos contienen los números de bloque de disco mismos. Es posible que el último bloque no esté lleno, así que el sistema no tiene forma de conocer el tamaño exacto de un archivo hasta el último byte (es decir, registra los tamaños de archivo en bloques, no en bytes).

21.1.5.2 Directorios en MS-DOS

Consideremos ahora algunos ejemplos de sistemas con árboles de directorios jerárquicos. La Fig. 5-12 muestra una entrada de directorio de MS-DOS, la cual tiene 32 bytes de longitud y contiene el nombre de archivo, los atributos y el número del primer bloque de disco. El primer número de bloque se emplea como índice de una tabla del tipo de la de la Fig. 5-9. Siguiendo la cadena, se pueden encontrar todos los bloques.

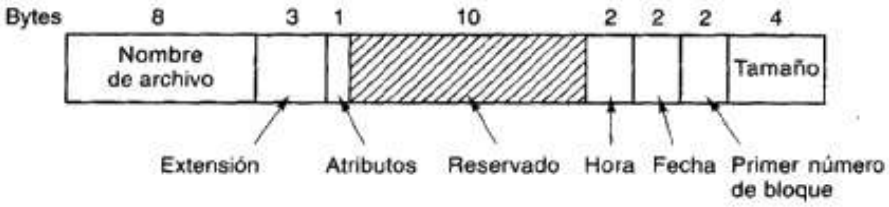


Figura 5-12. La entrada de directorio de MS-DOS.

En MS-DOS, los directorios pueden contener otros directorios, dando lugar a un sistema de archivos jerárquico. En este sistema operativo es común que los diferentes programas de aplicación comiencen por crear un directorio en el directorio raíz y pongan ahí todos sus archivos, con objeto de que no haya conflictos entre las aplicaciones.

21.1.5.3 Directorios en UNIX

La estructura de directorios que se usa tradicionalmente en UNIX es en extremo sencilla, como se aprecia en la Fig. 5-13. Cada entrada contiene sólo un nombre de archivo y su número de nodo-i. Toda la información acerca del tipo, tamaño, tiempos, propietario y bloques de disco está contenida en el nodo-i.
Algunos sistemas UNIX tienen una organización distinta, pero en todos los casos una entrada de directorio contiene en última instancia sólo una cadena ASCII y un número de nodo-i.
Cuando se abre un archivo, el sistema de archivos debe tomar el nombre que se le proporciona y
localizar sus bloques de disco. Consideremos cómo se busca el nombre de ruta /usr/ast/mbox. Usaremos UNIX como ejemplo, pero el algoritmo es básicamente el mismo para todos los sistemas de directorios jerárquicos. Lo primero que hace el sistema de archivos es localizará directorio raíz. En UNIX su nodo-i está situado en un lugar fijo del disco.
A continuación, el sistema de archivos busca el primer componente de la ruta, usr, en el directorio raíz para encontrar el número de nodo-i del archivo /usr. La localización de un nodo-itj

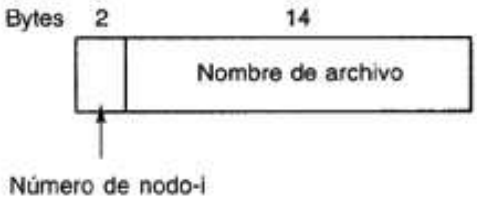


Figura 5-13. Una entrada de directorio UNIX.

una vez que se tiene su número es directa, ya que cada uno tiene una posición fija en el disco. Con la ayuda de este nodo-i, el sistema localiza el directorio correspondiente a./usr y busca el siguiente componente, ast, en él. Una vez que el sistema encuentra la entrada para ast, obtiene el nodo-i del directorio /usr/ast. Con este nodo, el sistema puede encontrar el directorio mismo y buscar mbox. A continuación se trae a la memoria el nodo-i de este archivo y se mantiene ahí hasta que se cierra el archivo. El proceso de búsqueda se ilustra en la F'g. 5-14.
Los nombres de ruta relativos se buscan de la misma forma que los absolutos, sólo que el punto de partida es el directorio de trabajo en lugar del directorio raíz. Cada directorio tiene entradas para . y .., que se colocan ahí cuando se crea el directorio. La entrada . tiene el número de nodo-i del directorio actual, y la entrada .. tiene el número de nodo-i del directorio padre. Así, un procedimiento que busque ../dick/prog.c simplemente buscará.. en el directorio de trabajo, encontrará el número de nodo-i de su directorio padre y buscará dick en ese directorio. No se requiere ningún mecanismo especial para manejar estos nombres. En lo que al sistema de directorios concierne, se trata de cadenas ASCII ordinarias, lo mismo que los demás nombres.

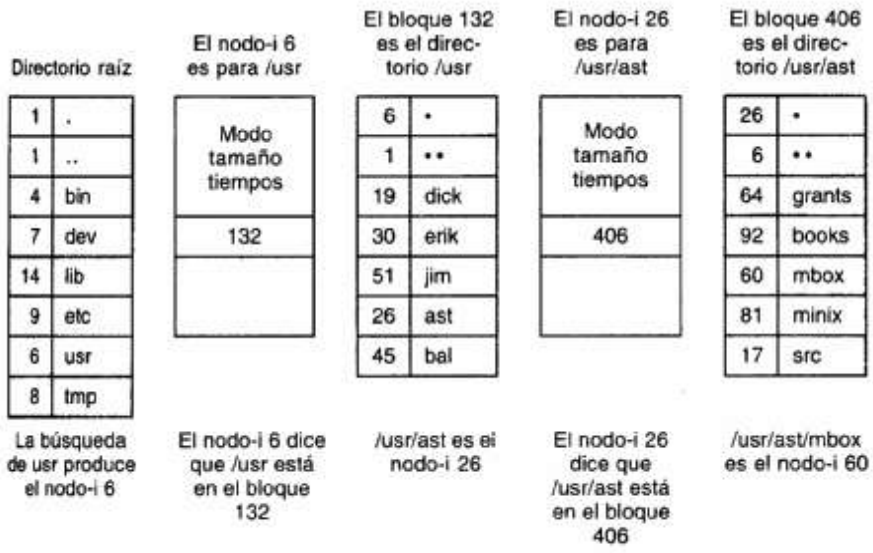


Figura 5-14. Pasos para buscar */usr/ast/mbox*.

21.1.6 Administración del espacio en disco:

Los archivos normalmente se almacenan en disco, así que la administración del espacio en disco es de interés primordial para los diseñadores de sistemas de archivos. Hay dos posibles estrategias para almacenar un archivo de n bytes: asignar n bytes consecutivos de espacio en disco, o dividir el archivo en varios bloques (no necesariamente) contiguos. Un trueque similar (entre segmentación pura y paginación) está presente en los sistemas de administración de memoria. El almacenamiento de un archivo como secuencia contigua de bytes tiene el problema obvio de que, si el archivo crece, probablemente tendrá que pasarse a otro lugar del disco. El mismo problema ocurre con los segmentos en la memoria, excepto que el traslado de un segmento en la memoria es una operación relativamente rápida comparada con el traslado de un archivo de una posición en el disco a otra.
Por esta razón, casi todos los sistemas de archivos dividen los archivos en bloques de tamaño fijo que no necesitan estar adyacentes.

22 Protección

Los diversos procesos de un sistema operativo se deben proteger de las actividades de los demás procesos. Existen mecanismos que garantizan que solo los procesos autorizados por el SO, puedan operar con los distintos recursos.

La *protección* se refiere a un mecanismo para controlar el acceso de procesos o usuarios al sistema, controles sobre ellos, etc. La *seguridad* es una medida de confianza que tenemos en que se preservará la integridad del sistema.

22.1.1 Objetivos de la protección

Originalmente se concibió para que los usuarios compartieran recursos sin peligro en sistemas multiprogramados. Actualmente evolucionaron para aumentar la confiabilidad de sistemas que usan recursos compartidos.

Razones para dar protección:

- Evitar que usuarios malintencionados violen restricciones de acceso.
- Asegurar que cada componente de un programa use los recursos respetando las políticas de uso de ellos.

La protección puede detectar errores en las interfaces entre subsistemas, evitando la contaminación de un subsistema sano, con otro que no funciona correctamente. Existen mecanismos para distinguir entre accesos autorizados y no autorizados.

Tipos de políticas de uso de recursos:

- Determinadas por el diseño del sistema
- Formuladas por los administradores del sistema
- Definidas por usuarios para proteger sus propios archivos o programas

Las políticas pueden cambiar en el tiempo o dependiendo de la aplicación, por lo tanto el sistema debe proveer de herramientas al programador para que pueda modificarlas o utilizarlas.

22.1.2 Dominios de protección

Un sistema consta de procesos y objetos, éstos últimos tanto de hardware (memoria, periféricos, etc.), como de software (archivos, semáforos, etc.). Cada objeto tiene un identificador, y se accede a él con operaciones bien definidas que dependen del objeto (CPU: Ejecución, Memoria: lectura, escritura, etc.).

Un proceso debe acceder a los recursos para los cuales está autorizado, y en un momento dado solo puede acceder a aquellos que necesite para llevar a cabo su tarea. Esto es conocido como el principio de necesidad de conocer.

22.1.2.1 Estructura de dominios

Un proceso opera dentro de un *dominio de protección*, que especifica los recursos a los que puede acceder. El dominio define el conjunto de objetos, y el tipo de operaciones que se pueden invocar sobre ellos. La capacidad para ejecutar una operación con un objeto, se llama *derecho de acceso* y un dominio, es una colección de derechos de acceso de la forma:

<nombre_de_objeto, conjunto_de_derechos>

Tipos de asociación entre proceso y dominio:

- **Estática:** El conjunto de recursos de un proceso no cambia durante la ejecución del mismo, y queremos respetar el principio de necesidad de conocer, por lo tanto necesitamos un mecanismo para cambiar el contexto de un dominio, para reflejar los derechos de acceso mínimos necesarios. Por ejemplo: Un proceso tiene dos fases. En una requiere escritura y en otra lectura. Si proporcionamos al proceso un dominio que incluye los dos accesos, proporcionamos más derechos de los necesarios a cada una de las fases.

- **Dinámica:** Se cuenta con un mecanismo que cambia de un dominio a otro. También se podría modificar el contenido del dominio, o en su defecto, crear un nuevo dominio con el contenido deseado, y cambiando a él.

Formas de establecer dominios:

- **Usuario:** El conjunto de objetos a los que se puede acceder depende de la identidad del usuario. Hay conmutación de dominio cuando se cambia de usuario (cierre y apertura de sesión).
- **Proceso:** El conjunto de objetos a los que se puede acceder depende de la identidad del proceso. La conmutación de dominio corresponde a que un proceso envía un mensaje a otro proceso y espera su respuesta.
- **Procedimiento:** El conjunto de objetos a los que se puede acceder corresponde a variables locales definidas dentro del procedimiento. La conmutación de dominio ocurre cuando se invoca a otro procedimiento.

22.1.2.2 Ejemplos

El modo dual (usuario – monitor) es una forma de proteger el SO. Cada uno tiene su propio dominio. En un sistema multiusuario no alcanza con esta protección puesto que se debe proteger un usuario de otro.

22.1.2.2.1 UNIX

Asocia un dominio a cada usuario, y cambiar de dominio significa cambiar temporalmente la identificación de usuario.

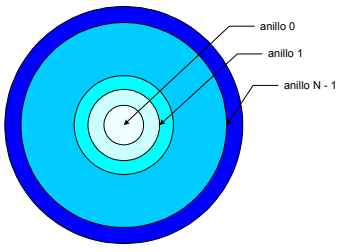
Cada archivo tiene asociados una identificación de propietario y un bit de dominio. Cuando un usuario con user-id = A, comienza a ejecutar un archivo de propiedad de B, con el bit de dominio apagado, el user-id del proceso se hace igual a A. Si el bit de dominio está prendido, el user-id del usuario se hace igual al del propietario del archivo, hasta que el proceso termina.

Esto se usa para poner a disposición e usuarios, los recursos privilegiados. Por ejemplo, el usuario puede tomar momentáneamente el user-id “root” para usar un recurso, pero hay que controlar que no pueda crear un archivo con user-id “root” y el bit encendido, porque podría convertirse en “root”.

Hay sistemas más restrictivos que no permiten cambiar los identificadores de usuario. Se deben usar mecanismos especiales para acceder a recursos especiales como “demonios”.

22.1.2.2.2 Multics

Los dominios de protección se organizan jerárquicamente en una estructura de anillos. Cada anillo corresponde a un dominio. El proceso que se ejecuta en el dominio D_0 es el que más privilegios tiene.



Tiene un espacio de direcciones segmentado donde cada segmento es un archivo y está asociado a un anillo. A cada proceso se le asocia un número de anillo, y puede acceder a segmentos asociados a procesos con menos privilegios (mayor número de anillo), pero no a los segmentos más privilegiados.

Se produce una conmutación de contexto cuando un proceso invoca un procedimiento de otro anillo, y esa conmutación es controlada por el SO. Para este control, el anillo cuenta con los campos:

- **Intervalo de acceso:** (b_1, b_2) y $b_1 \leq b_2$.
- **Limite:** b_3 tal que $b_3 > b_2$
- **Lista de puertas:** Identifica puertas en las que se pueden invocar segmentos.

Un proceso en el anillo i, puede ejecutar segmentos de un intervalo de acceso (b_1, b_2), solo si $b_1 \leq i \leq b_2$. Si no se cumple se efectúa una trampa y se hacen nuevos controles:

- $i < b_1$: Se permite la llamada porque el anillo que busco es menos privilegiado.
- $i > b_2$: Solo se permite la llamada si $b_3 \leq i$. La llamada se destina a una "puerta", permitiendo que los procesos con derechos de acceso limitado, puedan invocar procedimientos más privilegiados, pero de manera controlada.

Desventaja: No permite el cumplimiento de la necesidad de conocer.

Los mecanismos de protección también dependen de la necesidad de seguridad o performance. En una computadora en la que se procesan calificaciones de estudiantes, y a la que acceden estudiantes, se justifica un complejo sistema de seguridad, pero en una que solo es necesaria para triturar números, tiene más peso la performance.

22.1.3 Matriz de acceso

Las decisiones de política relativas a la protección se pueden implementar con una matriz de acceso. Las filas representan dominios, y las columnas objetos. Cada entrada es un conjunto de derechos de acceso.

Dominio \ Objeto	F_1	F_2	F_3	Impresora
D_1	leer		escribir	
D_2		leer		imprimir

Todos los procesos que se ejecutan en el dominio D_1, pueden leer el archivo F_1, y escribir el F_3. Los que se ejecutan en D_2, pueden leer el archivo F_2, e imprimir.

22.1.3.1.1 Conmutación

Si incluimos también a la matriz como un objeto, podemos darle privilegios solo a algunos dominios para modificarla. De la misma forma, podemos controlar la conmutación de dominios.

Dominio \ Objeto	F_1	F_2	F_3	Impresora	D_1	D_2
D_1	leer		escribir			conmutar
D_2		leer		imprimir		

Un proceso en el dominio D_1, puede conmutar al dominio D_2.

22.1.3.1.2 Copia

Se puede dar la capacidad de copiar derechos, añadiendo un (*) al derecho de acceso. Se puede copiar el derecho de acceso solo dentro de la misma columna (en el mismo objeto).

Variantes:

- **Transferencia:** Se copia el acceso (j, k) al acceso (i, k) y se borra el acceso (j, k).
- **Limitación:** El derecho (R*) se copia de (j, k) a (i, k) creando el derecho R, y no R*, por lo tanto un proceso en el dominio l, no puede copiarlo.

También el sistema podría elegir que derecho darle a la copia, si el de "copia", "trasferencia" o "copia limitada" colocando diferentes valores en la matriz.

Se necesita un mecanismo que permita añadir o quitar derechos. Para eso se le asigna el derecho de dueño a uno de los dominios que utiliza el objeto y solo él pede asignar o quitar derechos.

Se requiere un mecanismo que cambie derechos en las filas, y es el de "control".

22.1.4 Implementación de la matriz de acceso

La mayor parte de las entradas de la matriz están vacías.

22.1.4.1 Tabla global

Consiste en un conjunto de tripletas ordenadas:

<dominio, objeto, conjunto_de_derechos>

Si ejecuto una operación M, con un objeto O_j, en el dominio D_i, busco la entrada $<D_i, O_j, R_k>$ $M \in R_k$. Si no existe, se genera error.

Desventajas: La tabla es grande, por lo tanto no puede estar en memoria principal. Además hay información repetida, ya que si todos los dominios usan de la misma forma un mismo objeto, debo tener una entrada de la tabla para cada dominio.

22.1.4.2 Lista de acceso para objetos

Cada columna (objeto) es una lista de acceso, que contiene los pares *<dominio, conjunto_de_derechos>* (conjunto_de_derechos <> ∅).

En este caso se puede implementar un conjunto de derechos por omisión, que contiene todo los objetos a los que puedo acceder desde cualquier dominio.

22.1.4.3 Lista de capacidades para dominios

Es igual que en el caso anterior, pero para los dominios. Cada dominio tiene una lista de capacidades (objetos y operaciones). El usuario puede acceder a estas listas indirectamente, ya que están protegidas por el sistema.

Protección de capacidades:

- Cada objeto tiene una etiqueta para denotar su tipo (puntero, booleano, valor no inicializado). No son directamente accesibles al usuario.
- El espacio de direcciones de un programa se puede dividir:
    - Una parte accesible con datos e instrucciones normales.
    - Otra parte con la lista de capacidades accesibles

22.1.4.4 Un mecanismo de cerradura y llave

Es un término medio entre las dos listas (capacidades y acceso). Cada objeto tiene una lista de patrones de bit llamada *cerradura*, y cada dominio tiene otra llamada *llave*.

Para que un proceso que se ejecuta en un dominio pueda acceder a un objeto, el dominio debe tener la llave adecuada.

22.1.4.5 Comparación

Cuando un usuario crea un objeto, establece sus derechos, pero es difícil determinar el conjunto de derechos para cada dominio. Las listas de capacidades son útiles para localizar información de un proceso en particular pero no se corresponden con las necesidades del usuario.

El mecanismo de cerradura y llave, es un término medio. Puede ser eficaz y flexible, dependiendo de la longitud de las llaves, y los privilegios de acceso se pueden revocar para un dominio, simplemente cambiando su llave.

La mayor parte de los sistemas usa una combinación entre listas de acceso y capacidades.

22.1.5 Revocación de derechos de acceso

Tipos:

- **Inmediata o diferida**
- **Selectiva o general:**
    - *General:* Al revocar un derecho de acceso afecto a todos los usuarios con ese derecho.
    - *Selectiva:* Puedo seleccionar usuarios para revocarlo.

- **Parcial o total:**
  - *Total:* Revoco todos los derechos de acceso a un objeto.
  - *Parcial:* Revoco un subconjunto.
- **Temporal o permanente:**
  - *Temporal:* Se puede revocar el derecho y luego obtenerlo nuevamente.
  - *Permanente:* Si lo revoco, no puedo volverlo a obtener.

Con un sistema de lista de acceso la revocación es fácil. Es inmediata y puede ser general, selectiva, etc. Con listas de capacidades debo hallarlas en todo el sistema antes de revocarlas.

Esquemas de revocación:

- **Readquisición:** Periódicamente se eliminan capacidades de cada dominio. El proceso tiene que tratar de readquirirla para usarla, y si fue revocada, no la puede adquirir.
- **Retropunteros:** Cada objeto tiene punteros asociados a sus capacidades. Se puede seguir esos punteros para revocar o modificar capacidades. Su implementación es costosa.
- **Indirección:** Las capacidades no apuntan directamente al objeto sino a una entrada única de una tabla global. Para revocar, busco la entrada en la tabla global y la elimino. No permite revocación selectiva.
- **Claves:** La clave se define al crear la capacidad y el proceso que posee dicha capacidad no puede modificarla. Debo cambiar la clave maestra de cada objeto con un "set_key" para revocar los derechos. No permite revocación selectiva.

22.1.6 Sistemas basados en capacidades

22.1.6.1 Hydra

Es flexible. Proporciona un conjunto fijo de posibles derechos de acceso que el sistema conoce, y proporciona a un usuario del sistema de protección, un mecanismo para declarar nuevos derechos. Las operaciones sobre objetos, son tratadas también como objetos, y se accede a ellas a través de capacidades.

Otro concepto interesante es la *amplificación de derechos* que certifica un procedimiento como *confiable* y le permite acceder a objetos inaccesibles, a nombre de otro procedimiento que si puede accederlos. Estos derechos no son universales, y están restringidos.

Ej: Un procedimiento invoca una operación P, la cual es amplificada. La operación utiliza sus derechos nuevos, y cuando termina, el procedimiento vuelve a tener los derechos normales.

Hydra diseñó protección directa contra el problema de *subsistemas que sospechan uno del otro*. Cuando varios usuarios invocan un programa de servicios, asumen el riesgo de que falle, perdiendo datos, o que retenga los datos y puedan ser usados posteriormente sin autorización.

22.1.6.2 Sistema Cambridge CAP

Es más sencillo que Hydra. Proporciona también protección segura a datos de usuario.

Capacidades:

- **Capacidad de datos:** Sirven para obtener acceso a objetos y tienen los derechos estándar (leer, escribir, etc.). Está en el micro código de CAP.
- **Capacidad de software:** Corre por cuenta de un procedimiento privilegiado. Cuando es invocado, él mismo obtiene los derechos de leer y escribir.

22.1.7 Protección basada en el lenguaje

La protección se logra con la ayuda del núcleo del SO que valida los intentos de acceso a recursos. El gasto de inspeccionar y validar todos los intentos de acceso a todos los recursos es muy grande, por lo tanto debe ser apoyada por hardware.

Al aumentar la complejidad del SO, se deben refinar los mecanismos de protección. Los sistemas de protección, no solo se preocupan de si puedo acceder a un recurso, sino también de cómo lo accedo, por lo tanto los diseñadores de aplicaciones deben protegerlos, y no solo el SO.

Los diseñadores de aplicaciones mediante herramientas de los lenguajes de programación pueden declarar la protección junto con la tipificación de los datos.

Ventajas:

- Las necesidades de protección se declaran sencillamente y no llamando procedimientos del SO.
- Las necesidades de protección pueden expresarse independientemente de los recursos que ofrece el SO.
- El diseñador no debe proporcionar mecanismos para hacer cumplir la protección.
- Los privilegios de acceso están íntimamente relacionados con el tipo de datos que se declara.

Diferencias entre las distintas formas de protección:

- **Seguridad:** La obligación de cumplimiento por núcleo ofrece un grado de seguridad que el código de seguridad ofrecido por el compilador.
- **Flexibilidad:** La flexibilidad de la implementación por núcleo es limitada. Si un lenguaje no ofrece suficiente flexibilidad, se puede extender o sustituir, perturbando menos cambios en el sistema que si tuviera que modificarse el núcleo.
- **Eficiencia:** Se logra mayor eficiencia cuando el hardware apoya la protección.

La especificación de protección en un lenguaje de programación permite describir en alto nivel las políticas de asignación y uso de recursos.

El programador de aplicaciones necesita un mecanismo de control de acceso seguro y dinámico para distribuir capacidades a los recursos del sistema entre los procesos de usuario.

Las construcciones que permiten al programador declarar las restricciones tienen tres operaciones básicas:

- Distribuir capacidades de manera segura y eficiente entre procesos clientes.

- Especificar el tipo de operaciones que un proceso podría invocar en un recurso asignado.
- Especificar el orden en que un proceso dado puede invocar las operaciones de un recurso.

23 Seguridad

La protección es estrictamente un problema interno, la seguridad en cambio no solo requiere un sistema de protección adecuado, sino también considerar el entorno externo donde el sistema opera.

La protección no es útil si la consola del operador está al alcance de personal no autorizado. La información debe protegerse de accesos no autorizados, destrucción, alteración mal intencionada o inconsistencias.

23.1.1 El problema de la seguridad

Un sistema es seguro si los recursos se usan y acceden como es debido en todas las circunstancias. Esto no siempre es posible pero se debe contar con mecanismos que garanticen que las violaciones de seguridad sean un suceso poco común.

Formas de acceso mal intencionado:

- Lectura no autorizada de datos (robo de información).
- Modificación no autorizada de datos.
- Destrucción no autorizada de datos.

El hardware del sistema debe proporcionar protección que permita implementar funciones de seguridad. MS-DOS casi no ofrece seguridad, y sería relativamente complejo agregar nuevas funciones.

23.1.2 Validación

Se deben identificar los usuarios del sistema.

23.1.2.1 Contraseñas

Cuando el usuario se identifica con un identificador, se le pide una contraseña, y si ésta coincide con la almacenada en el sistema, se supone que el usuario está autorizado. Se usan cuando no se cuenta con sistemas de protección más completos.

23.1.2.2 Vulnerabilidad de las contraseñas

El problema es la dificultad de mantener secreta una contraseña. Para averiguarla se puede usar fuerza bruta, usando todas las combinaciones posibles de letras, números y signos de puntuación, o probando con todas las palabras del diccionario.

El intruso podría estar observando al usuario cuando la digita o “husmeando” en la red, o sea, viendo los datos que el usuario transfiere a la red, incluidos identificadores y contraseñas.

Si el sistema elige la contraseña, puede ser difícil de recordar, y el usuario la anota, si la elige el usuario, puede ser fácil de adivinar.

Algunos sistemas envejecen las contraseñas, obligando al usuario a cambiarlas cada determinado tiempo. Se controla que el usuario no vuelva a usar una contraseña que ya usó antes, guardando un historial de contraseñas.

23.1.2.3 Contraseñas cifradas

El sistema UNIX usa cifrado para no tener que mantener en secreto su lista de contraseñas. Cada usuario tiene una contraseña y el sistema contiene una función, extremadamente difícil de invertir, pero fácil de calcular, que codifica las contraseñas y las almacena codificadas. Cuando el usuario presenta la contraseña, se codifica y compara con las ya codificadas. De esta forma no es necesario mantener en secreto el archivo con contraseñas.

Problemas:

- Se puede obtener una copia del archivo de contraseñas y comenzar a cifrar palabras hasta descubrir una correspondencia (ya que el algoritmo de cifrado de UNIX es conocido). Las nuevas versiones de UNIX ocultan el archivo.
- Muchos sistemas UNIX solo tratan los primeros 8 caracteres como significativos, quedando pocas opciones de contraseña.
- Algunos sistemas prohíben el uso de palabras de diccionario como contraseña, para que sea más difícil de adivinar.

23.1.3 Contraseñas de un solo uso

Se pueden usar contraseñas algorítmicas, donde el sistema y el usuario comparten un secreto. El sistema proporciona una semilla, el usuario con el algoritmo y la semilla introduce la contraseña, y el sistema la comprueba (ya que conoce el algoritmo del usuario). La contraseña es de un solo uso, por lo tanto, cualquiera que la capte, no podría utilizarla luego.

23.1.4 Amenazas por programas

23.1.4.1 Caballo de Troya

Muchos sistemas permiten que un usuario pueda ejecutar programas de otro usuario. Estos programas podrían aprovechar los derechos que le proporciona el dominio del usuario ejecutante y abusar de ellos. Un segmento de código que abusa de su entorno es llamado *caballo de Troya*.

23.1.4.2 Puerta secreta (Trap door)

El diseñador del programa podría dejar una puerta, por la cual entra salteándose todos los protocolos de seguridad. Se podría generar una puerta ingeniosa en un compilador, que coloque puertas en todos los programas que compile.

Una puerta en un sistema es difícil de detectar por la cantidad de código que el sistema posee.

23.1.5 Amenazas al sistema

23.1.5.1 Gusanos

El gusano engendra copias de si mismo ocupando los recursos del sistema, impidiendo que sean usados por otros procesos. En las redes son muy potentes, ya que pueden reproducirse entre los sistemas, y paralizar toda la red.

23.1.5.1.1 Gusano de Morris:

Causó en 1988 pérdidas millonarias a través de Internet. Era auto-replicante, para distribuirse rápidamente, además características de UNIX, le permitieron propagarse por todo el sistema. El gusano aprovechó defectos en rutinas de seguridad de UNIX, para obtener accesos no autorizados a miles de sitios interconectados. Ver ejemplo de página 631. Interesante pero inútil.

23.1.5.2 Virus

También están diseñados para extenderse hacia otros programas, y pueden producir daños graves en un sistema, como modificar o destruir archivos y causar caídas en programas o sistemas. Un gusano es un programa completo y autónomo; un virus es un fragmento de código incrustado en un programa legítimo.

Los computadores multiusuario no son tan vulnerables, porque protegen los programas ejecutables contra escritura, aún así es fácil su infección.

Antivirus: La mayor parte solo actúan sobre virus conocidos, examinando los programas del sistema en busca de instrucciones específicas que sabe que contiene el virus.

23.1.6 Vigilancia de amenazas

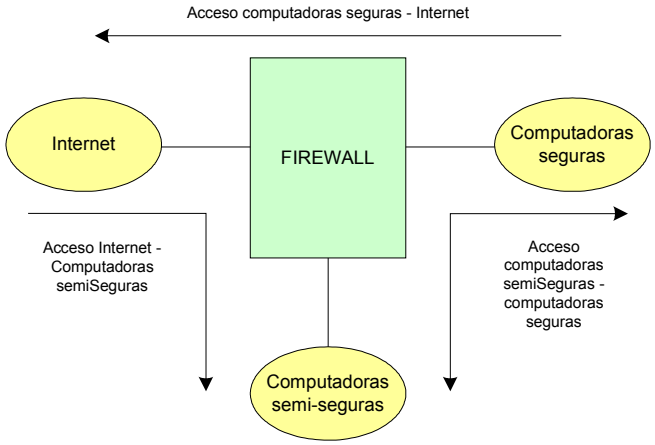
Se buscan patrones de actividad sospechosos. Las técnicas son:

- Cuenta de las veces que se proporcionan contraseñas incorrectas en el login
- Llevar una *bitácora de auditoría* que registra la hora, el usuario y los accesos a objetos, para determinar los problemas, pero estas bitácoras pueden crecer mucho, ocupando demasiados recursos en el sistema.
- Explorar el sistema periódicamente en busca de “agujeros” de seguridad, que se pueden realizar en el computador cuando hay poco tráfico, no como en el caso de las bitácoras. Puede examinar:

- Contraseñas fáciles de adivinar.
- Programas setuid no autorizados (si el sistema maneja el mecanismo).
- Programas no autorizados en directorios del sistema.
- Procesos cuya ejecución tiene una duración inusual.
- Protecciones inapropiadas de directorios de usuario y sistema.
- Protecciones inapropiadas de archivos de datos, contraseñas, drivers, o el núcleo del sistema.
- Caballos de troya.

Los computadores en red son susceptibles a ataques por la gran cantidad de ataques remotos posibles.

Una ayuda de protección en red son los firewalls. Un firewall es un computador que se coloca entre lo confiable y lo no confiable, limita el acceso por red, y lleva una bitácora.



23.1.7 Cifrado

Por la red se transmite gran cantidad de información que no se desea que sea descubierta. Para lograr esto, se sigue el siguiente mecanismo:

- La información se cifra (codifica) de su forma comprensible original (texto limpio) a una forma interna (texto cifrado). La forma interna se puede leer pero no es comprensible.
- El texto cifrado se puede almacenar en archivos legibles y enviarse por canales desprotegidos.
- Para entender el texto, el receptor debe descifrarlo (decodificarlo) para obtener el texto limpio.

El reto es crear esquemas de cifrado imposibles de romper.

Hay varios métodos, pero uno de ellos consiste en tener un algoritmo de cifrado general E, uno de descifrado general D, y una o más claves secretas.

E_k y D_k son los algoritmos para la clave k.

Para el mensaje m se debe cumplir:

1. $D_k(E_k(m)) = m$
2. D_k, E_k se calculan de forma eficiente.
3. Se debe mantener en secreto la clave y no los algoritmos.

Se creó el DES, estándar de cifrado de datos, que contiene el problema de distribución de claves, ya que hay que enviar secretamente las claves.

Una solución a esto es usando clave pública. Cada usuario tiene dos claves, una pública y una privada, y para comunicarse solo basta con saber la pública del otro.

23.1.7.1.1 Algoritmo:

Clave pública = (e, n)
Clave privada = (d, n)

El mensaje se representa con un entero y se divide en submensajes de largo 0.. (n – 1)

$E(m) = m^e \bmod n = C$
$D(C) = C^d \bmod n$

n se calcula como el producto de dos primos p y q al azar (n = p x q); d es al azar, y satisface mcd [d, (p-1) x (q-1)] = 1; e se calcula tal que e x d mod [(p-1) x (q-1)] = 1.

Si es necesario, ver ejemplo en página 638.

23.1.8 Clasificación de seguridad de los computadores

El total de sistemas de computación que obligan al cumplimiento correcto de políticas de seguridad son llamados *Base de computador confiable (TCB)*.
Se clasifican en A, B, C y D:

- **D – Protección mínima:** Son sistemas evaluados que no satisfacen los requisitos de seguridad de las otras clases. Ej: MS-DOS y Windows 3.1.
- **C – Protección discrecional:** Se contabilizan los usuarios y sus acciones mediante auditorías. Tiene dos niveles:
  - **C1:** Permite que los usuarios puedan proteger información privada y evitar que otros la lean o destruyan. Abarca la mayor parte de las versiones de UNIX. La TCB de C1 controla el acceso entre usuarios y archivos permitiendo compartir objetos entre individuos nombrados, o grupos definidos. El usuario debe identificarse antes de iniciar cualquier actividad que la TCB deba mediar.
  - **C2:** Une a los requisitos de C1, un control de acceso a nivel individual, y el administrador del sistema puede auditar de forma selecta las acciones de cualquier usuario conociendo su identidad. Hay algunas versiones de UNIX que certifican el nivel C2.
- **B – Protección obligatoria:** Tienen todas las propiedades de C2, y además "pegan" etiquetas de confidencialidad a cada objeto.
  - **B1:** Mantiene la etiqueta de seguridad en cada objeto del sistema la cual es usada para tomar decisiones relacionadas con el control de acceso obligatorio. Maneja niveles de seguridad jerárquicos, de manera que usuarios pueden acceder a cualquier objeto con etiqueta de confidencialidad igual o menor.
  - **B2:** Extiende etiquetas de confidencialidad a los recursos del sistema. Se asignan a los dispositivos físicos, para seleccionar los usuarios que pueden usarlos.
  - **B3:** Permite crear listas de control de acceso que denotan usuarios a los que no se concede acceso a un objeto nombrado dado. La TCB contiene un mecanismo que detecta violaciones de seguridad, avisando al administrador y finalizando los sucesos.

- **A – Nivel alto de protección:** Funciona arquitectónicamente igual que B3, pero contiene técnicas de verificación con más alto grado que las especificadas por TCB.

23.1.9 Ejemplo de modelo de seguridad: Windows NT

Va desde seguridad mínima (por defecto) hasta C2. El administrador puede configurarlo al nivel deseado mediante C2config.exe. El modelo se basa en el concepto de "cuentas de usuario". Permitiendo crear cualquier cantidad de cuentas de usuario que se pueden agrupar de cualquier manera.

Cuando un usuario ingresa al sistema, NT crea un *testigo de seguridad* que contiene todos los datos y permisos del usuario, y que va a ser accedido por los procesos cada vez que el usuario quiere acceder a un objeto. La validación de una cuenta de usuario se efectúa a través del nombre de usuario y contraseña.

NT cuenta con una auditoría integrada y permite vigilar amenazas de seguridad comunes. Los atributos de seguridad de un objeto NT se describen con un descriptor de seguridad que contiene el identificador de seguridad del propietario del objeto (quien puede cambiar los permisos), un identificador de seguridad de grupo, una lista de control de acceso discrecional (usuarios o grupos a los que se le permite o prohíbe el acceso) y una lista de control de acceso al sistema (guarda mensajes de auditoria del sistema).

NT clasifica los objetos como contenedores o no contenedores. Los contenedores pueden contener lógicamente otros objetos (Ej: Directorios).

El "subobjeto" hereda los permisos del padre. Si se cambian los del padre, cambian los del hijo.

24 El sistema UNIX

24.1.1 Historia

La primera versión fue en 1969 en Bell Laboratories, y MULTICS tuvo una gran influencia sobre él. La tercer versión fue el resultado de escribir la mayor parte del SO en lenguaje C, en lugar de lenguaje ensamblador, agregando funciones de multiprogramación.
La primera versión liberada fue la 6, en 1976, y se utilizó en universidades.
Con el pasar del tiempo se fueron agregando funciones de memoria virtual, paginación y memoria compartida. Se apoyaron los protocolos TCP/IP, logrando gran popularidad y permitiendo un masivo crecimiento de Internet. Se implementó una nueva interfaz con el usuario (C-shell), un nuevo editor de textos (vi), y compiladores.
En 1986 se liberó la versión 4.3BSD, sobre la que se basa este capítulo.
El conjunto actual de sistemas UNIX no está limitado por los producidos por Bell Laboratories, por lo tanto también existen una gran cantidad de interfaces de usuario que los distintos proveedores distribuyen.
UNIX está disponible para casi todos los computadores de propósito general, como PC's, estaciones de trabajo, etc. en entornos que van de académicos a militares.
Se trata de crear un standard para lograr una única interfaz y popularizar aún mas el sistema.

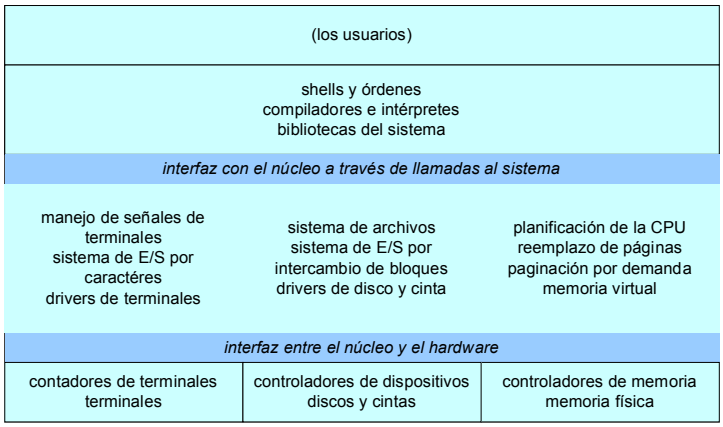
24.1.2 Principios de diseño

UNIX se diseñó como un sistema de tiempo compartido. El shell es sencillo y puede ser sustituido por otro si se desea. El sistema de archivos es un árbol con múltiples niveles, y las dependencias y peculiaridades de los dispositivos se mantienen en el núcleo hasta donde es posible, confinadas en su mayor parte a los drivers de dispositivos.
UNIX apoya múltiples procesos. La planificación de la CPU es un sencillo algoritmo de prioridades. Se usa paginación por demanda para gestión de memoria, usando un área de SWAP si hay hiperpaginación.
El sistema es sumamente flexible, y cuenta con recursos para ampliarlo fácilmente. Al ser diseñado para programadores, cuenta con recursos como *make* (útil para compilar archivos fuentes de un programa) y el *Sistema de control de código fuente* (SCCS) para mantener versiones sucesivas de archivos sin necesidad de almacenarlos. Está escrito en C, por lo tanto es portable.

24.1.3 Interfaz con el programador

Consta de dos partes separables, el núcleo y los programas de sistemas.

Está formado por capas, y los servicios del SO son proporcionados a través de llamadas al sistema. Las llamadas al sistema definen la *interfaz con el programador*, y el conjunto de programas del sistema, la *interfaz con el usuario*, las dos apoyadas por el núcleo.
Las llamadas al sistema se agrupan en tres categorías explicadas a continuación.



24.1.3.1 Manipulación de archivos

Un *archivo* en UNIX es una secuencia de bytes. El núcleo no impone ninguna estructura de archivo, aunque los diferentes programas esperan distintos niveles de estructura. Están organizados en *directorios* con estructura de árbol. Los directorios también son archivos con información para encontrar otros archivos. Un *nombre de camino* (path) consiste en nombres de archivo (directorios) separados por '/', donde la base de partida es el directorio root.

UNIX tiene:

- **Nombres de camino absoluto:** Parten de la raíz del sistema y se distinguen por comenzar con una diagonal. Ej. "/usr/local".
- **Nombres de camino relativo:** Parten del directorio actual.

Maneja enlaces:

- **Simbólicos o blandos:** Son archivos que contienen el nombre de camino de otro archivo. Pueden apuntar a directorios y cruzar fronteras de sistemas de archivo (a diferencia de los duros).
- **Duros:**
  - El nombre del archivo '.' es un enlace duro al directorio mismo.
  - El nombre del archivo '..' es un enlace duro al directorio padre.

Los dispositivos de hardware tienen nombres en el sistema de archivos. Los directorios comunes son:

- **/bin:** Contiene binarios de los programas UNIX.
- **/lib:** Contiene bibliotecas.
- **/user:** Contiene los directorios de usuarios.
- **/tmp:** Contiene archivos temporales.
- **/etc.**: Contiene programas administrativos.

Para trabajar con archivos y directorios uso las siguientes llamadas al sistema:

- **creat:** Crea un archivo:
- **open, close:** Abre y cierra archivos.
- **read, write:** Trabaja sobre el archivo.
- **link, unlink:** Crea o destruye enlaces duros a archivos.
- **symlink:** Crea un enlace simbólico al archivo.

- **mkdir, rmdir:** Crea o elimina directorios.
- **cd:** Cambia de directorio.

24.1.3.2 Control de procesos

Los procesos se identifican con su *identificador de proceso* (PID), y se crea un proceso nuevo con la llamada al sistema **fork**.
El proceso nuevo consiste en una copia del espacio de direcciones del padre, y ambos continúan su ejecución en la instrucción siguiente al **fork**, con la diferencia que el código de retorno del **fork** es 0 para el hijo, y es el PID del hijo para el padre.
El proceso termina con una llamada al sistema **exit**, y el padre puede esperarlo con **wait**.
El hijo al terminar queda como *difunto* por si el padre necesita su información. En el caso que el padre termine, sin esperar por el hijo, el hijo pasa a ser un proceso *zombie*.
Todos los procesos de usuario son hijos de un proceso original llamado *init*.
Tipos de identificador de usuario:

- **Efectivo:** Se usa para los permisos de acceso a archivos, y se hace igual al id del usuario dueño del archivo, para abrirlo.
- **Real:** Es el id del usuario.

Esto permite a ciertos usuarios tener privilegios más allá de los ordinarios, pero controlados por el SO.

24.1.3.3 Señales

Son un recurso para manejar excepciones, y son similares a las interrupciones de software.

- **SIGINT:** Interrupción. Detiene una orden antes de que termine de ejecutarse. Se produce con el carácter ^C.
- **SIGQUIT:** Abandonar. Detiene un programa y vacía su memoria en un archivo llamado core (^bs).
- **SIGILL:** Instrucción no válida.
- **SIGSEV:** Acceso inválido a memoria.
- **SIGKILL:** No se puede pasar por alto. Sirve para terminar un proceso desbocado que no hace caso a otras señales como SIGINT o SIGQUIT.

No hay prioridades entre señales, y si mando dos del mismo tipo a un proceso, se sobrescriben. También existen señales para entregar datos urgentes en una red, o dormir procesos hasta que sean interrumpidos.

24.1.3.4 Grupos de procesos

Grupos relacionados de procesos podrían cooperar para realizar una tarea común, y es posible enviar señales a todos los procesos del grupo.
Solo un grupo de procesos puede usar una terminal de E/S en un momento dado. Este trabajo de *primer plano* tiene la atención del usuario de la terminal, mientras los trabajos de *segundo plano* realizan otras tareas.

24.1.3.5 Manipulación de información

Existen llamadas al sistema para devolver y ajustar temporalizadores, hora actual, y para obtener el PID de un proceso (getpid), nombre de una máquina (gethostname), etc.

24.1.3.6 Rutinas de biblioteca

La interfaz de llamadas al sistema de UNIX se apoya con bibliotecas y archivos de cabecera. Estos últimos proporcionan estructuras de datos complejas que se emplean en las llamadas. Por ejemplo <stdio.h> proporciona una interfaz que lee y escribe miles de bytes usando buffers locales. Hay otras para funciones matemáticas, acceso a redes, etc.

24.1.4 Interfaz con el usuario

Los programadores y usuarios manejan los programas de sistema que UNIX provee. Estos efectúan llamadas al sistema que en muchos casos no son obvias para el usuario.

- **Programa ls:** Lista los nombres de archivos del directorio actual. Hay opciones:
  - **- l:** Listado largo. Muestra el nombre de archivo, dueño, protección, fechas y tamaño.
- **Programa cp:** Copia un archivo existente.
- **Programa mv:** Cambia un archivo de lugar o nombre.
- **Programa rm:** Borra archivos
- **Programa cat:** Muestra el contenido de un archivo en la pantalla.

También hay editores (vi), compiladores (C, PASCAL), programas para comparar (cmp, diff), buscar patrones (grep), etc.

24.1.4.1 Shells y órdenes

Los programas normalmente se ejecutan con un intérprete de órdenes, que en el caso de UNIX es un proceso de usuario como cualquier otro llamado shell. Hay diferentes shells, pero todos tienen órdenes en común.
El shell indica que está listo exhibiendo una señal de espera (prompt) que depende del tipo de shell (ej.: en shell C es "%"), y la orden de usuario es de una única línea.
Generalmente el shell espera (wait) hasta que terminó la tarea que el usuario ordenó, pero se pueden ejecutar *tareas en segundo plano* donde la tarea queda ejecutándose y el shell sigue interpretando órdenes.
El shell C cuenta con un *control de trabajos* que permite transferir procesos entre el primer y el segundo plano.

24.1.4.2 E/S estándar

Los procesos pueden abrir archivos a discreción, pero siempre tienen abiertos tres descriptores de archivo en su inicio:

- **0:** Entrada estándar.
- **1:** Salida estándar.
- **2:** Error estándar.

Y éstos, les permite trabajar con ellos (ej: pantalla, teclado, etc.).
Esos archivos son por defecto, pero puedo asignar otros si es necesario (ej: impresora como salida). Este cambio es llamado *redirección de E/S.*

24.1.4.3 Conductos, filtros y guiones de shell

Se usa un conducto para llevar datos de un proceso a otro. En UNIX se usa una barra vertical '|' (pipe), como por ejemplo "ls | lpr" que imprime lo que devolvió el comando ls.
Una orden que pasa su entrada estándar a su salida estándar realizando un proceso sobre ella, es llamada *filtro*.
Un archivo que contiene órdenes de shell es un *guión de shell* (shell script). Se ejecuta como cualquier orden y se invoca el shell apropiado para leerlo.

24.1.5 Gestión de procesos

ES fácil crear y manipular procesos en UNIX. Los procesos se representan con bloques de control, y se almacenan en el núcleo, el cual los usa para controlar el proceso y planificar la CPU.

24.1.5.1 Bloques de control de procesos

La *estructura de proceso* contiene lo que el sistema necesita para saber acerca de él.
Existe un arreglo de estructuras de proceso, cuyo tamaño se define al enlazar el sistema.
El planificador implementa la cola de procesos listos como una lista doblemente enlazada de estructuras e proceso.
El *espacio de direcciones virtual* del proceso se divide en segmentos de texto (código del programa), datos y pila. Los dos últimos están en el mismo espacio de direcciones, pero como son variables crecen en direcciones opuestas.
Las *tablas de páginas* registran información de la correspondencia entre memoria virtual y física.
En la *estructura de usuario* se guardan los id de usuario y grupo, el directorio actual, la tabla de archivos abiertos, etc.
Cuando un proceso se ejecuta en modo de sistema, se utiliza una pila de núcleo para el proceso, en lugar de la pila del proceso. La pila de núcleo y la estructura de usuario constituyen el *segmento de datos del sistema* de ese proceso.
El **fork** copia los datos y la pila en el nuevo proceso, al contrario del **vfork**.

24.1.5.2 Planificación de CPU

Está diseñada para beneficiar los procesos interactivos. Cada proceso tiene una *prioridad de planificación* (a mayor número, menor prioridad). Los procesos que realizan tareas importantes tienen prioridades menores que "pzero" y no se pueden matar. Los demás tienen prioridades positivas.
Cuanto más tiempo de CPU acumula un proceso, más baja su prioridad, y se usa envejecimiento para evitar inanición.
En el núcleo, ningún proceso desaloja a otro.

24.1.6 Gestión de memoria

24.1.6.1 Intercambio

Un proceso más grande que la memoria no podía ser ejecutado, y los segmentos de datos de sistema y usuario, que se mantenían en memoria principal, causaban fragmentación externa.
La asignación de memoria virtual y espacio de intercambio es por primer ajuste. Si la memoria aumentaba se buscaba un nuevo hueco y se copiaba. Si no existía tal hueco, se cambiaba a disco.
El *proceso planificador* decide que procesos se deben intercambiar, y se despierta cada 4 segundos para revisar. Para evitar hiperpaginación, no se intercambia a disco un proceso que no haya estado un cierto tiempo en memoria.
Todas las particiones de swap deben tener el mismo tamaño (potencia de dos) y deben poder accederse con diferentes brazos del disco.

24.1.6.2 Paginación

Utiliza paginación por demanda, eliminando la fragmentación externa, y reduciendo la cantidad de intercambios.
Algunas arquitecturas, no cuentan con un bit de referencia a página en hardware, impidiendo el uso de muchos algoritmos de gestión de memoria. Utiliza una modificación del algoritmo de segunda oportunidad. La escritura de páginas se efectúa en cúmulos para mejorar el desempeño.
Se realizan verificaciones para que el número de páginas en memoria para cada proceso no se reduzca demasiado, o no aumente demasiado. El encargado es *pagedaemon* que está dormido hasta que el número de marcos libres se reduce más allá de un umbral.
En VAX el tamaño de página es pequeño causando ineficiencias de la E/S, así que se juntan en grupos de a dos. El tamaño de página efectivo no necesariamente es idéntico al tamaño de página de hardware, aunque si debe ser múltiplo.

24.1.7 Sistema de archivos

24.1.7.1 Bloques y fragmentos

El sistema de archivos está ocupado por *bloques de datos* los cuales deben ser mayores a 512 bites, por razones de velocidad, pero no muy grandes por causa de la fragmentación interna.
UNIX usa dos tamaños de bloque. Todos los bloques de un archivo son grandes menos el último que es llamado *fragment*. Los tamaños se establecen al crear el sistema de archivos, obligando que la proporción bloque-fragmento sea 8:1.
Se debe conocer el tamaño del archivo antes de copiarlo a disco, a fin de saber si debo copiar un bloque o un fragmento. De otra forma tendría que copiar 7 fragmentos, y luego pasarlos a un bloque, causando ineficiencia.

24.1.7.2 I-nodos

Un archivo se representa con un *i-nodo* que es un registro que guarda información acerca del archivo. Contiene:

- Identificadores de usuario y grupo.
- Hora de última modificación y último acceso.
- Cuenta del nnúmero de enlaces duros.
- Tipo de archivo.
- 15 punteros a bloques
    - **12 primeros:** Apuntan a *bloques directos*, o sea son una dirección de un bloque de datos.
    - **3 siguientes:** Apuntan a *bloques indirectos*.
        - **Primero:** Apunta a un *bloque de indirección simple*, que es un bloque índice con punteros a bloques de datos.
        - **Segundo:** Es un puntero a un *bloque de indirección doble*, el cual apunta a un bloque índice que contiene direcciones de bloques índices que apuntan a bloques de datos.
        - **Tercero:** Es un puntero a un *bloque de indirección triple*. En UNIX no se usa, ya que el tamaño de los archivos puede alcanzar solo 232 bytes (por ser un sistema de direccionamiento de 32 bits), y para ello alcanza con el segundo.

24.1.7.3 Directorios

Se representan igual que los archivos, pero difieren en el campo “tipo” del i-nodo, además que el directorio tiene una estructura específica.
Los primeros nombres de cada directorio son ‘.’ y ‘..’. Para buscar un archivo, voy recorriendo paso a paso el camino, y no todo junto, ya que en muchos casos, cuando cambio de dispositivo, también cambia el sistema de archivos. El último nombre de un camino, debe ser el de un archivo.
Debido a la existencia de enlaces, para evitar ciclos, en una búsqueda no puedo pasar por más de 8 enlaces.
Los archivos que no están en disco (dispositivos) no tienen bloques asignados. El núcleo los detecta y llama a los controladores de E/S apropiados.

24.1.7.4 Transformación de un descriptor de archivo en un i-nodo

Para trabajar con un archivo, debo pasar un descriptor de él como argumento a las llamadas al sistema. El núcleo lo usa como entrada de una tabla que apunta al i-nodo del archivo. La *tabla de archivos abiertos* tiene una longitud fija, por lo tanto no puedo abrir archivos ilimitadamente.

24.1.7.5 Estructuras de disco

Los dispositivos físicos se dividen en dispositivos lógicos, y cada uno de estos últimos define un sistema de archivos físico, para diferentes usos. Una de las particiones es la de swap.
El primer sector del dispositivo lógico es el *bloque de arranque* que puede contener un programa de autoarranque primario, que llama al secundario que está en los próximos 7,5 Kb.

24.1.7.6 Organización y políticas de asignación

El núcleo usa un par <*número de dispositivo lógico, número de i-nodo*> para identificar un archivo. No puedo asegurar la confiabilidad del sistema de archivos, ya que se mantiene en memoria, escribiéndolo a disco cada 30 segundos. Por lo que un corte de energía podría producir inconsistencia.
Se introdujo el sistema de *grupo de cilindros* para localizar bloques de archivos logrando casi no tener que mover la cabeza del disco. Cada cilindro guarda información útil para recuperación. Siempre se trata que todos los bloques de un archivo estén dentro del mismo grupo de cilindros.

24.1.8 Sistemas de E/S

Se trata de ocultar las peculiaridades de los dispositivos, y se hace en el núcleo con el *sistema de E/S*, que consiste en un sistema de buffers en caché, y drivers para los dispositivos (que conoce las peculiaridades de ellos).
Hay tres tipos principales de dispositivos:

- **Por bloques:** Discos y cintas. Se pueden direccionar directamente en bloques de tamaño fijo. Las transferencias se hacen a través del *caché de buffers de bloques*.
- **Por caracteres:** Terminales, impresoras, etc. Agrupa los dispositivos que no usan caché de buffers de bloques, como por ejemplo las interfaces de gráficos de alta velocidad.
- **Interfaz de socket**

Un dispositivo se distingue por:

- **Tipo**
- **Número de dispositivo:** Se divide en:
    - **Número de dispositivo principal:** Indica el lugar en el arreglo de dispositivo, para encontrar puntos de ingreso al driver apropiado.
    - **Número de dispositivo secundario:** Es interpretado por el driver, como por ejemplo una partición lógica de disco, etc.

24.1.8.1 Caché de buffers de bloques

Consiste en varias cabeceras de buffer que pueden apuntar a memoria física, a un número de dispositivo, o a un número de bloque en el dispositivo.
Los buffers no usados se mantienen en tres listas:

- **Recientemente usados:** Enlazados en orden LRU (Lista LRU).
- **No usados recientemente:** Sin contenido válido (Lista AGE).
- **Vacíos:** No tienen memoria física asociada (Lista EMPTY).

Al requerir una lectura, se busca en el caché de buffers. Si se encuentra, se usa, y si no se encuentra, se lee del dispositivo, y se guardan los datos en el caché, eligiendo un buffer de la lista AGE, o si está vacía, de la lista LRU.
La cantidad de datos del buffer es variable, y el tamaño mínimo es el tamaño del cúmulo de paginación. El tamaño del caché, tiene un efecto considerable en el desempeño del sistema, dependiendo de él, la cantidad de transferencias de E/S que se deben realizar.
Cuando escribo en una cinta, debo sincronizar los buffers, para que escriban en un orden determinado. Cuando escribo en un disco, los puedo sincronizar para lograr el menor movimiento posible de la cabeza del disco.

24.1.8.2 Interfaces con dispositivos crudas

Se les llama a las interfaces por caracteres que tienen los dispositivos por bloques, y lo que hacen
es pasar por alto el caché de bloques. En estos casos, el tamaño de la transferencia está limitado por los dispositivos físicos, algunos de los cuales requieren un número par de bytes.

24.1.8.3 Listas C

Los controladores de terminales usan un sistema de buffer de caracteres, que mantiene bloques pequeños de caracteres en listas enlazadas. Tanto las salidas como entradas se controlan con interrupciones.
Los editores de pantalla completa y otros programas que necesitan reaccionar con cada digitación, usan el *modo crudo* que pasa por alto el buffer de caracteres.

24.1.9 Comunicación entre procesos (IPC)

La comunicación entre procesos no es uno de los puntos fuertes de UNIX. En muchos casos no se permite la *memoria compartida* porque algunas arquitecturas no apoyaban su uso. En los actuales si, pero de todas formas, presenta problemas en un entorno de red, ya que los accesos por la red no son tan rápidos como los accesos locales, perdiendo la ventaja de velocidad en la memoria compartida.

24.1.9.1 Sockets

Es el mecanismo de IPC característico de UNIX. Un *conducto* permite establecer un flujo de bytes unidireccional entre dos procesos, y se implementan como archivos ordinarios (con algunas excepciones). El tamaño es fijo, y si esta lleno, los procesos que intentan escribir en él, se suspenden. Como ventaja, es pequeño, y su escritura no pasa por disco, sino a través de los buffers de bloques.
En 4.3BSD se implementan con sockets, que proporcionan una interfaz general, tanto en la misma máquina, como en redes. Puede ser usado entre procesos no relacionados entre si.
Los socket tienen vinculadas *direcciones*, cuya naturaleza depende del *dominio de comunicación del socket.* Dos procesos que emplean el mismo dominio, tienen el mismo formato de direcciones. Los tres dominios que se implementan son el dominio de UNIX (AF_UNIX), cuyo formato de direcciones es el de los nombres de caminos del sistema de archivos (ej: /casa/auto), el dominio de Internet (AF_INET) y el de Servicios de red (AF_NS).
Hay varios tipos de sockets, los cuales se clasifican dependiendo del servicio que realizan, y el recurso socket tiene un conjunto de llamadas específicas al sistema. Luego de establecer una conexión por sockets, se conocen las direcciones de ambos puntos, y no se necesita ningún tipo de información para transferir datos. Algunos tipos de socket, no conocen conexiones (datagramas), y se les debe especificar el destinatario del mensaje.

24.1.9.2 Soporte para redes

Reconoce gran cantidad de protocolos de Internet, y el núcleo se diseñó con miras de facilitar la implementación de protocolos adicionales.
Los procesos de usuario se comunican con los protocolos de red a través del recurso de sockets; y los sockets se apoyan con protocolos, los cuales prestan sus servicios.
Suele haber un *driver de interfaz de red* para cada tipo de controlador de red, que maneja las características específicas de la red local, asegurando que los protocolos no tengan que preocuparse por ello. Las funciones de la interfaz de red, dependen del hardware de red, y de las funciones que él apoya (como trasmisión segura, etc.).

25 El sistema Linux

25.1.1 Historia

Es muy parecido a UNIX y tiene la compatibilidad fue una de las metas de su diseño. Su desarrollo se inició en 1991, para el 80386 (de 32 bits).
El código fuente se ofrecía gratuitamente en Internet, por lo tanto su historia ha sido una colaboración de muchos usuarios de todo el mundo.
Se divide entre el *núcleo* y un *sistema* Linux. El núcleo fue desarrollado de cero por la comunidad Linux, y el sistema incluye gran cantidad de componentes. Es un entorno estándar para aplicaciones y programación, y una *distribución* Linux incluye todos los componentes estándar, junto con un conjunto de herramientas administrativas que simplifican la instalación y desinstalación de paquetes, etc.

25.1.1.1 El núcleo de Linux

La primer versión, en 1991 no trabajaba con redes y sólo se ejecutaba con procesadores compatibles con el 80386, además, el soporte de drivers era limitado. No tenía soporte para archivos con correspondencia en memoria virtual, pero manejaba páginas compartidas. Solo reconocía un único sistema de archivos (Minix), pero implementaba correctamente procesos UNIX.
En 1994, salió la siguiente versión que apoyaba el trabajo con redes, con los protocolos TCP/IP y UNIX, sockets, y nuevos drivers de dispositivos (para redes o Internet). También incluía nuevos sistemas de archivos y reconocía controladores SCSI. El subsistema de memoria virtual se extendió para apoyar el intercambio por paginación, pero de todas formas seguía restringido a la plataforma Intel. Además se implementó la comunicación entre procesos (IPC) de UNIX, con memoria compartida, semáforos y cola de mensajes.
En 1994 salió la versión 1.2, soportando nuevo hardware, y la arquitectura de bus PCI. Se añadió soporte para emular el sistema DOS. Incluía soporte para otros procesadores, aunque no completo.
En 1996 la versión 2.0 Soportaba múltiples arquitecturas (inclusive la Alpha de 64 bits) y arquitecturas multiprocesador. Se mejoró la gestión de memoria, y se introdujo soporte de hilos internos del núcleo y carga automática de módulos por demanda.

25.1.1.2 El sistema Linux

El sistema Linux es código común a varios sistemas operativos tipo UNIX. Las bibliotecas principales se comenzaron como GNU, pero se fueron mejorando y agregando funcionalidades. El sistema Linux es mantenido por una red de desarrolladores que colaboran por Internet, pero se tiene un libro de normas, para asegurar la compatibilidad.

25.1.1.3 Distribuciones de Linux

Incluyen el sistema Linux básico, utilitarios de instalación y administración, paquetes, herramientas, procesadores de texto, editores, juegos, etc. facilitando la tarea de la persona que lo quiere instalar en su máquina, y poniendo cada componente en su lugar apropiado. Hay distribuciones comerciales y no comerciales como Red Hat, SuSE, Mandrake, etc.

25.1.1.4 Licencias de Linux

El núcleo se distribuye bajo la Licencia Publica General de GNU. Los autores renunciaron a los derechos de autor sobre el software, pero los derechos de código siguen siendo propiedad de dichos autores. Linux es software gratuito, y nadie que haga un derivado, lo puede registrar como propio.

25.1.2 Principios de Diseño

Linux es un sistema multiusuario, multitarea con un conjunto completo de herramientas compatibles con UNIX. Linux desde sus principios procuró sacar el máximo de funcionalidad posible en recursos limitados, por lo tanto puede ejecutarse en muchos sistemas pequeños.

25.1.2.1 Componentes de un sistema Linux

El sistema Linux se divide en tres partes principales de código:

- **Núcleo:** Mantiene las abstracciones importantes del sistema operativo como memoria virtual y procesos. Implementa todas las operaciones necesarias para calificar como SO.
- **Bibliotecas del sistema:** Definen un conjunto estándar de funciones a través de las cuales las aplicaciones pueden interactuar con el núcleo y que implementan gran parte de la funcionalidad del SO, que no necesita todos los privilegios del código de núcleo.
- **Utilitarios del sistema:** Son programas que realizan tareas de administración. Implementa por ejemplo los “demonios” que son procesos que se ejecutan de forma permanente.

El código de núcleo se ejecuta en el modo privilegiado del procesador (*modo de núcleo*) con pleno acceso a todos los recursos físicos. No se incorpora código del modo usuario en el núcleo, el cual es colocado en bibliotecas del sistema.

El núcleo de Linux se crea como algo separado, aislado (estructura monolítica) al igual que UNIX. Este sistema tiene muchas ventajas, dado que el núcleo se mantiene en un solo espacio de direcciones, no se requieren conmutaciones de contexto cuando un proceso invoca a una función del SO. Además dentro de ese espacio de direcciones se encuentra el código de planificación, memoria virtual central, todo el código de núcleo, incluido el de los drivers de dispositivos, sistema de archivos y trabajo con redes.

El sistema operativo que el núcleo de Linux ofrece no se parece en absoluto a un sistema UNIX. Faltan muchas características extra de UNIX, y las funciones que si se ofrecen no son necesariamente del formato que utiliza UNIX. El núcleo no mantiene una interfaz con el SO que ven las aplicaciones, mas bien las aplicaciones para comunicarse con el SO deben hacer llamadas a las bibliotecas del sistema las cuales solicitan servicios del SO.

Las bibliotecas del sistema ofrecen funcionalidad, permitiendo hacer solicitudes de servicio al núcleo, y otras rutinas como funciones matemáticas, etc.

Los programas en modo usuario, incluyen programas necesarios para iniciar el sistema, configurar la red, programas de inicio de sesión de usuarios, etc. También incluye operaciones sencillas como listar directorios, trabajar con archivos, y otras más complejas como procesadores de texto, etc.

25.1.3 Módulos del Núcleo

El núcleo de Linux tiene la facultad de cargar y descargar módulos del núcleo cuando se le pide hacerlo. Estos se ejecutan en modo de núcleo privilegiado, teniendo acceso a todas las capacidades de hardware. Los módulos pueden implementar desde un driver hasta un protocolo de redes.

La ventaja de esta modularización, es que cualquiera puede variar a su gusto algunos de los módulos núcleo (usualmente no es necesario re-compilar todo el núcleo, después que se hizo una variación en algún modulo del sistema). Otra ventaja es que se puede configurar el sistema con un núcleo mínimo, y agregar controladores mientras se usa, como por ejemplo el montaje de un CD-ROM, etc.

El soporte de módulos en Linux tiene tres componentes principales:

- **Gestión de módulos:** Permite cargar módulos en la memoria y comunicarse con el resto del sistema.
- **Registro de controladores:** Permite a los módulos notificar al resto del núcleo que está disponible un nuevo controlador.
- **Resolución de conflictos:** Permite a los drivers reservar recursos de hardware y protegerlos.

25.1.3.1 Gestión de módulos

La carga de un módulo no es solo la carga en memoria del núcleo, sino que el sistema debe asegurarse de que cualquiera referencia que el módulo haga a símbolos o puntos de ingreso del núcleo se actualicen de modo que apunten a las posiciones correctas dentro del espacio de direccionamiento del núcleo. Si el sistema no puede resolver alguna referencia del módulo, el módulo se rechaza.
La carga se efectúa en dos etapas:

- Se reserva un área continua de memoria virtual del núcleo para el módulo, y se devuelve un puntero a esa dirección.
- Se pasa el módulo junto con la tabla de símbolos al núcleo, copiando el módulo sin alteraciones en el espacio previamente asignado.

Otro componente de la gestión de módulos encontramos el solicitador de módulos, que es una interfaz de comunicación con la cual puede conectarse un programa de gestión de

módulos. La función que cumple es de avisar cuando el proceso necesite un driver, o sistema de archivo etc., que no este cargado actualmente, y dará la oportunidad de cargar ese servicio.

25.1.3.2 Registro de controladores

El núcleo mantiene tablas dinámicas de todos los controladores conocidos, y proporciona un conjunto de rutinas que permite añadir o quitar controladores de de estas tablas en cualquier momento. Las tablas de registro incluyen los siguientes elementos:

- **Drivers de dispositivos:** Incluyen los dispositivos de caracteres (impresoras, teclados), dispositivos por bloques (Disco) y dispositivos de interfaz de red.
- **Sistemas de archivos:** Puede ser cualquier cosa que implemente rutinas de invocación del sistema de archivos virtual de Linux.
- **Protocolos de red:** Puede implementar un protocolo de red completo o un conjunto de reglas de un firewall.
- **Formato binario:** Especifica una forma de reconocer, y cargar, un nuevo tipo de archivo ejecutable.

25.1.3.3 Resolución de conflictos

Linux ofrece un mecanismo central de resolución de conflictos que ayuda a arbitrar el acceso a ciertos recursos de hardware. Sus objetivos son:

- Evitar que los módulos entren en conflictos al tratar de acceder a recursos de hardware.
- Evitar que los sondeos de drivers que auto detectan la configuración de un dispositivo interfieran con los drivers de dispositivos existentes.
- Resolver conflictos entre múltiples drivers que tratan de acceder al mismo hardware.

Para esto el núcleo mantiene una lista de recursos de hardware asignados. Por lo que si cualquier controlador de dispositivo quiere acceder a un recurso, debe reservar primero el recurso en la lista del núcleo. Si la reservación se rechaza, debido a que esta en uso, corresponde al módulo decidir como proceder.

25.1.4 Gestión de procesos

25.1.4.1 El modelo de proceso fork/exec

A diferencia de UNIX, en Linux cualquier programa puede invocar **execve** en cualquier momento sin necesidad de crear un proceso nuevo. Bajo Linux la información que contiene un proceso se puede dividir en varias secciones específicas, descriptas a continuación.

25.1.4.1.1 Identidad del proceso

Consiste en los siguientes elementos:

- **Identificador del proceso (PID):** Es único y sirve para especificar procesos al SO. También puede tener identificadores adicionales de grupo.
- **Credenciales:** Cada proceso debe tener asociado un identificador de usuario y uno o mas identificadores de grupo, que determinan los derechos de acceso a lo recursos y archivos del sistema que el proceso tiene.
- **Personalidad:** Cada proceso tiene asociado un identificador de personalidad el cual sirve para modificar un poco las semánticas de ciertas llamadas al sistema. Sirven para crear compatibilidad con algunos sistemas UNIX.

22.4.1.2 Entorno del proceso

Se hereda del padre y consiste en dos vectores terminados con el carácter nulo:

- **Vector de argumentos:** Lista los argumentos de línea de órdenes que se usan para invocar el programa que se esta ejecutando (por convención comienza con el nombre del programa del mismo).
- **Vector de entorno:** Es una lista de pares "NOMBRE = VALOR" que asocia variables de entorno con nombre a valores de texto arbitrarios.

22.4.1.3 Contexto de un proceso

La identidad del proceso no se cambia en su ejecución. El contexto es el estado del programa en ejecución en cualquier instante dado, y cambia constantemente. Se divide en:

- **Contexto de planificación:** Contiene la información que el planificador necesita para suspender y reiniciar el proceso. Se incluye copias guardadas de todos los registros del proceso. También incluye información acerca de la prioridad de planificación, y de señales pendientes que esperan ser entregadas al proceso.
- **Contabilidad:** El núcleo mantiene información acerca de los recursos que cada proceso está consumiendo, y el total de recursos que el proceso ha consumido durante toda su existencia.
- **Tablas de archivos:** Es un arreglo de punteros a estructuras de archivos del núcleo. Al efectuarse una llamada al sistema para E/S con archivos, los procesos se refieren a los archivos empleando su índice en esta tabla.
- **Contexto del sistema de archivos:** Guarda los nombres de los directorios raíz y por omisión que se usarán para buscar archivos.
- **Tabla de manejadores de señales:** Define la rutina invocara cuando lleguen señales específicas.
- **Contexto de memoria virtual:** Describe todo el contenido del espacio de direcciones privado de dicho proceso.

25.1.4.2 Procesos e Hilos

Dos hilos adentro de un proceso, a diferencia de los procesos, comparten el mismo espacio de direcciones.
El núcleo de Linux maneja de forma sencilla la creación de hilos y procesos, ya que utiliza exactamente la misma representación interna para todos. Un hilo no es más que un proceso nuevo que comparte el mismo espacio de direcciones que su padre. La distinción entre un proceso y un hilo se hace sólo cuando se crea un hilo nuevo con la llamada "clone". Mientras que "fork" crea un proceso nuevo que tiene su propio contexto totalmente nuevo, "clone" crea un proceso nuevo que tiene identidad propia pero que puede compartir las estructuras de datos de su padre.
La llamada al sistema "fork" es un caso especial de la llamada "clone" que copia todos los subcontextos y no comparte ninguno.

25.1.5 Planificación

En Linux la planificación no cumple solo con la función de ejecución e interrupción de procesos, sino que también tiene asociado la ejecución de diversas tareas del núcleo.

25.1.5.1 Sincronización del núcleo

La sincronización del núcleo implica mucho más que la simple planificación de procesos. Se requiere una estructura que permita la ejecución de las secciones críticas del núcleo sin que otra sección crítica la interrumpa, para que se puedan ejecutar concurrentemente las tareas del núcleo con las rutinas de servicio de interrupciones.

La primera técnica para solucionar el problema radica en hacer no expropiable el código de núcleo normal. Una vez que un fragmento del código del núcleo comienza a ejecutarse se puede garantizar que será el único código del núcleo que se ejecute hasta que pase algo de lo siguiente:

- **Interrupción:** Solo son un problema si sus rutinas de servicio contienen secciones críticas.
- **Fallo de página:** Si una rutina del núcleo trata de leer o escribir la memoria del usuario, esto podría incurrir en un fallo de página, que requiera E/S de disco y el proceso en ejecución se suspenderá hasta que termine la E/S.
- **Llamada al planificador desde el código del núcleo:** El proceso se suspenderá y habrá una replanificación.

En conclusión con esta técnica el código del núcleo nunca será desalojado por otro proceso, y no es necesario tomar medidas especiales para proteger las secciones críticas,

teniendo como precondición que las secciones críticas no tenga referencias a memoria de usuario ni esperas para que termine una E/S.

La segunda técnica que Linux utiliza para proteger a las secciones críticas es inhabilitando las interrupciones durante una sección critica, garantizando que podrá continuar sin que haya peligro de acceso concurrente a estructuras de datos compartidas. El problema de este sistema es que muy costoso.

Linux plantea una solución intermedia para poder aplicar secciones críticas largas sin tener que inhabilitar las interrupciones. Para esto divide en varias partes el servicio de interrupciones.

Manejador de interrupciones de la mitad superior
Manejador de interrupciones de mitad inferior
Rutinas de servicios del núcleo – sistema (no desalojables)
Programas en modo usuario (desalojables)

Cada nivel puede ser interrumpido por código que se ejecute en un nivel más alto, pero nunca por código del mismo nivel, o de un nivel inferior, con excepción del código en modo de usuario.

25.1.5.2 Planificación de procesos

Linux tiene dos algoritmos de planificación de procesos independientes:

- **Algoritmo de tiempo compartido para una planificación expropiativa justa entre procesos:** Es implementado con prioridad *basada en créditos*. Cada proceso tiene un cierto número de créditos de planificación, cuando es preciso escoger una nueva tarea para ejecutar se selecciona el proceso con mas créditos. Cada vez que se produce una interrupción del temporizador el proceso pierde un crédito, cuando llegan a cero se suspende y se escoge otro proceso. Además se tiene una forma de renovación de créditos en las que se añaden créditos a todos los procesos del sistema (crédito = crédito /2 + prioridad).
- **Algoritmo para tareas en tiempo real en las que las prioridades absolutas son más importantes que la equitatividad:** Con este sistema el procesa que entra primero es el proceso que tiene mas prioridad (utiliza una cola circular), si hay dos con la misma prioridad ejecuta el proceso que ha esperado mas tiempo. Este planificador es de tiempo real blando, ofrece garantías acerca de las prioridades pero no acerca de la rapidez.

25.1.5.3 Multiprocesamiento simétrico

El núcleo de Linux implementa correctamente el *multiprocesamiento simétrico* (SMP). La desventaja que tiene, es que procesos e hilos se pueden ejecutar en paralelo en distintos procesadores, pero no puede estar ejecutándose código del núcleo en más de un procesador.

25.1.6 Gestión de memoria

La gestión de memoria en Linux tiene dos componentes:

- **Sistema de gestión de memoria física:** Se encarga de asignar y liberar páginas, grupos de páginas y bloques pequeños de memoria.
- **Sistema de gestión de memoria virtual:** Maneja la memoria virtual, que es memoria que tiene una correspondencia con el espacio de direcciones de procesos en ejecución.

25.1.6.1 Gestión de memoria física

El administrador primario de memoria física es el asignador de páginas, encargado de asignar y liberar todas las páginas físicas además de asignar intervalos de páginas contiguas físicamente si se lo solicitan. El asignador utiliza el algoritmo de *montículo de compañeras* para saber las páginas que están disponibles. Si dos regiones compañeras asignadas quedan libres se combinan para formar una región más grande. Si se solicita una pequeña, y no hay, se subdivide recursivamente una grande, hasta lograr el tamaño deseado.
Las funciones del núcleo no tienen que usar el asignador básico para reservar memoria, existen otros subsistemas de gestión de memoria especializados, dentro de ellos encontramos:

- **Memoria Virtual.**
- **Asignador de longitud variable:** Utiliza el asignador de páginas *kmalloc,* el cual cumple solicitudes de tamaño arbitrario que no necesariamente se conocen con anticipación. Este asignador esta protegido contra interrupciones.
- **El caché de buffer:** Es el caché principal del núcleo para dispositivos orientados a bloques.
- **El caché de páginas:** Almacena páginas enteras del contenido de archivos, de todo tipo de dispositivos (no solo bloques).

Los tres sistemas interactúan entre ellos.

25.1.6.2 Memoria virtual

Se encarga de mantener el espacio de direcciones visible para cada proceso. Se crean páginas por solicitud. Mantiene dos vistas del espacio de direcciones:

- **Vista lógica:** Consiste en un conjunto de regiones que no se traslapan. Cada región se describe internamente como una sola estructura que define las propiedades (lectura, escritura, etc.). Las regiones se enlazan en un árbol binario balanceado.
- **Vista física:** Almacena las tablas de páginas de hardware para el proceso. Guardan la ubicación exacta de la página, que puede estar en disco o memoria. Cada dirección de la tabla guarda un puntero a una tabla de funciones que controla el espacio de memoria indicado.

Regiones de memoria virtual

Hay varios tipos dependiendo de las propiedades:

- **Almacenamiento auxiliar:** Describe el origen de las páginas de una región. Se puede respaldar en un archivo, o no respaldar (tipo más sencillo de memoria virtual).
- **Reacción a las escrituras:** La correspondencia entre una región y el espacio de direcciones del proceso puede ser *privada* o *compartida*.
    - **Privada:** Se requiere copiar al escribir para que los cambios sean locales al proceso.
    - **Compartida:** Se actualiza para que todos los procesos vean de inmediato la modificación.

Tiempo de vida de un espacio de direcciones virtual

Hay dos formas de crear procesos:

- **Exec:** Se crea un espacio de direcciones virtual nuevo.
- **Fork:** Se copia el espacio de direcciones de un proceso existente (el padre). Cuando debo copiar una región de correspondencia privada, las modificaciones que hagan el hijo y el padre son privadas, por lo tanto la página se debe marcar como "solo lectura", para que se deba realizar una copia antes de modificarla.

Intercambio y paginación

Los primeros sistemas UNIX intercambian a disco procesos enteros. Actualmente se intercambian páginas. El sistema de paginación se divide en dos secciones:

- **Algoritmo de política:** Decide que páginas pasar a disco y cuando escribirlas.
- **Mecanismo de paginación:** Transfiere la página a memoria cuando se la necesita.

Linux emplea un mecanismo modificado del algoritmo de segunda oportunidad.

Memoria virtual del núcleo

Linux reserva para el núcleo una región constante, que depende de la arquitectura, y las entradas de la tabla de páginas correspondiente a esa zona, se marcan como protegidas. Esta área contiene dos regiones:

- **Estática:** Contiene referencias de tabla de páginas, a cada página de memoria física disponible en el sistema, para traducir fácilmente direcciones físicas en virtuales cuando se ejecuta código del núcleo.
- **Sin propósito específico:** El núcleo en esta zona puede modificar entradas de tabla de páginas para que apunten donde quiera.

25.1.6.3 Ejecución y carga de programas de usuario

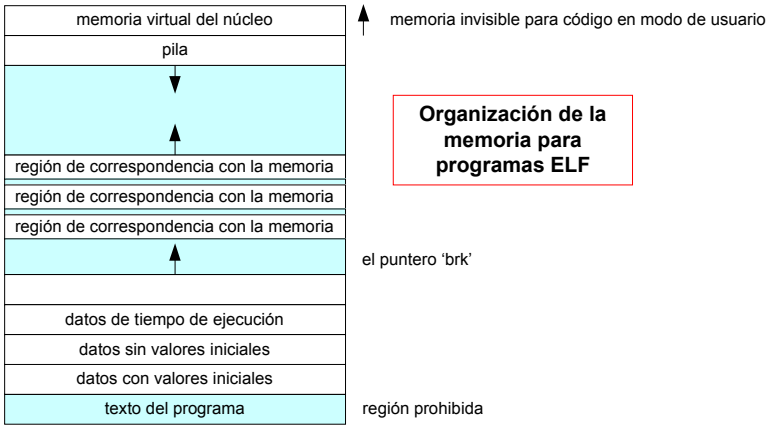
La ejecución se dispara con la llamada al sistema **exec**, que ordena ejecutar un programa nuevo dentro del proceso actual. Las tareas del SO son:

1. Verificar que el proceso invocador tenga permiso de hacerlo.

2. Invocar la rutina de carga para iniciar la ejecución del programa.

No hay una única rutina de carga sino que Linux mantiene una tabla de rutinas cargadoras, ya que en la evolución de Linux, se cambió el formato de los archivos binarios. Antiguamente el formato era .out, actualmente es ELF, que es más flexible y extensible.

Correspondencia entre los programas y la memoria



Al comenzar la ejecución, no se carga el programa en memoria física, sino que se establece la correspondencia entre las páginas del archivo binario, y las regiones de memoria virtual. Solo cuando el programa trata de acceder a una página, se genera un fallo de página, y se la trae a la memoria física.
El formato ELF consiste en una cabecera seguida de varias secciones alineadas con páginas. El cargador lee la cabecera y establece correspondencia entre las secciones del programa y la memoria virtual.

Enlazado estático y dinámico

Luego de cargar el programa, también se necesita cargar en memoria las funciones del sistema que utiliza este programa. En el caso más sencillo, estás se incrustan en el código binario (enlace estático) con el problema que se repite gran cantidad de código entre todos los programas. El enlace *dinámico* soluciona este problema.
Linux implementa el enlace dinámico a través de una biblioteca. Cuando enlazo el programa, se coloca en él un pequeño programa con instrucciones para cargar esas bibliotecas que necesita.
Las bibliotecas se compilan en código independiente de la posición (PIC) que se puede ejecutar en cualquier dirección de memoria.

25.1.7 Sistema de archivos

Usa el sistema de archivos estándar de UNIX. Un archivo no tiene que ser un objeto, sino que puede ser cualquier cosa capaz de manejar la entrada y salida de datos, como por ejemplo un driver.
A través de una capa de software, el *sistema de archivos virtual* (VFS) se encarga de abstraer sus peculiaridades.

25.1.7.1 El sistema de archivos virtual

Está orientado a objetos y tiene dos componentes:

- Definiciones sobre el aspecto del archivo. Los tres tipos principales de objetos son:
    - **i-nodo, objeto archivo:** Son mecanismos empleados para acceder a archivos.
        - **i-nodo:** Representa el archivo como un todo. Mantiene información estándar acerca del archivo.
        - **objeto archivo:** Representa un punto de acceso a los datos del archivo.
    - **Objeto sistema de archivos:** Representa un conjunto conectado de archivos que forma una jerarquía de directorios, y su principal obligación es dar acceso a los i-nodos.
- Capa de software para manejar dichos objetos.

Se define un conjunto de operaciones que cada uno de los tres tipos debe implementar, así la capa de software, puede realizar funciones sobre los objetos independientemente de su tipo.
Los archivos de directorio en Linux se manejan de una forma un tanto diferente a los demás archivos, por lo tanto tienen definidas operaciones específicas.

25.1.7.2 El sistema de archivos Linux ext2fs

Es similar al ffs (fast file system) de BSD. Los archivos de directorio se almacenan en disco, al igual que los archivos normales, pero su contenido se interpreta de diferente manera.
Ext2fs no usa fragmentos, sino que realiza todas sus asignaciones en unidades más pequeñas. El tamaño del bloque es de 1kb (aunque también se manejan bloques de 2 y 4 Kb). Está dividido en *grupos de bloques*, y dentro de ellos, el planificador trata de mantener las asignaciones físicamente contiguas, tratando siempre de reducir la fragmentación. Además la fragmentación se compensa con que los bloques están cercanos, por lo tanto no es necesario mover mucho la cabeza del disco para leerlos a todos.

25.1.7.3 El sistema de archivos proc de Linux

El *sistema de archivos de procesos* es un sistema de archivos cuyo contenido no está almacenado realmente en ninguna parte sino que se calcula bajo demanda, según las solicitudes de E/S de los usuarios.
Cada subdirectorio no corresponde a un directorio de disco, sino a un proceso activo del sistema, y el nombre del directorio es la representación ASCII del PID del proceso.
De esta forma se puede dar un listado de todos los procesos activos con solamente la llamada al sistema **ps**.

25.1.8 Entrada y salida

Los drivers de dispositivos son archivos, por lo tanto un usuario puede abrir un canal de acceso a un dispositivo de la misma forma que puede abrir cualquier otro archivo. Además, puede utilizar el mismo sistema de protección de archivos para proteger los dispositivos.
Linux divide todos los dispositivos en tres clases:

- **Dispositivos por bloques:** Suelen usarse para almacenar sistemas de archivos, pero también se permite el acceso directo a estos dispositivos para que los programas puedan crear y reparar el sistema de archivos que el dispositivo contiene.
- **Dispositivos por caracteres:** No necesitan apoyar toda la funcionalidad de los archivos normales. Por ejemplo, es útil manejar una búsqueda en un dispositivo de cinta, pero no en un mouse.
- **Dispositivos de red:** Se tratan de manera diferente. Los usuarios no pueden transferir datos directamente, sino que deben abrir una conexión.

25.1.8.1 Dispositivos por bloques

Para asegurar un rápido acceso a los discos se utiliza el *caché de buffers de bloques* y el *gestor de solicitudes*.

25.1.8.1.1 El caché de buffers por bloques

El caché de buffers sirve como reserva de buffers para E/S activa y como caché para la E/S finalizada.
El caché de buffers cuenta con dos partes:

- **Los buffers mismos:** Son un conjunto de páginas cuyo tamaño cambia dinámicamente y se asignan desde la reserva de memoria principal del núcleo. Cada página se divide en varios buffers de igual tamaño.
- **Los buffer_heads:** Son un conjunto de descriptores de buffers de caché. Contienen la información que el núcleo tiene acerca de los buffers (el dispositivo al que pertenece el buffer, la distancia entre los datos dentro de ese dispositivo, y el tamaño del buffer).

Los buffers son guardados en diferentes tipos de listas: para buffers limpios, sucios, bloqueados y libres. Hay dos demonios en segundo plano que ayudan a escribir los buffers sucios. Uno se despierta a intervalos regulares, y escribe los buffers que han estado sucios por un determinado tiempo. El otro es un hilo del núcleo que se despierta cuando hay una proporción demasiado alta del caché de buffers que está sucia.

25.1.8.1.2 El gestor de solicitudes

Es la capa de software que controla la lectura y escritura del contenido de los buffers de y a un driver de dispositivo por bloque.

Se mantiene una lista individual de solicitudes para cada controlador de dispositivo por bloques, y las solicitudes se planifican según el algoritmo C-SCAN.
A medida que se hacen nuevas solicitudes de E/S, el gestor intenta fusionar las solicitudes de las listas por dispositivo, pasando de varias solicitudes pequeñas, a una grande.
Se pueden hacer solicitudes de E/S que pasen por alto el caché de buffers, y se usa generalmente en el núcleo, y en memoria virtual.

25.1.8.2 Dispositivos por caracteres

El núcleo prácticamente no preprocesa las solicitudes de leer o escribir en un dispositivo por caracteres, y se limita a pasar la solicitud al dispositivo en cuestión para que él se ocupe de ella.

25.1.9 Comunicación entre Procesos

25.1.9.1 Sincronización y señales

Para informar a un proceso que sucedió un suceso se utiliza una *señal*. Este sistema tiene la desventaja de que el número de señales es limitado y estas no pueden enviar información, solo puede comunicar a un proceso el hecho de que ocurrió un suceso.
El núcleo de Linux no usa señales para comunicarse con los procesos que se están ejecutando en modo de núcleo. Generalmente hay una cola de espera para cada suceso, y cuando ese suceso se produce, despierta todos los procesos que están en esa cola.
Otra forma que utiliza Linux como comunicación son los semáforos.

25.1.9.2 Transferencia de datos entre procesos

Linux ofrece el mecanismo *pipe* (conducto) que permite a un proceso hijo heredar un canal de comunicación con su padre.
La memoria compartida ofrece un mecanismo extremadamente rápido para comunicar cantidades grandes o pequeñas de datos. Pero tiene como desventaja que no ofrece mecanismo de sincronización, por lo cual es recomendable usarla con algún otro método de sincronización.

25.1.10Estructura de redes

Linux apoya tanto los protocolos estándar de Internet de UNIX y de otros SO. El trabajo de redes se implementa con tres capas de software:

- **Interfaz de sockets:** Las aplicaciones de usuario efectúan todas las solicitudes de trabajo con redes a través de esta interfaz.
- **Controladores de protocolos:** Cada vez que llegan datos de trabajo con redes a esta capa, se espera que los datos estén etiquetados con un identificador que

indique el protocolo de red. Cuando termina de procesar los paquetes, los envía a la siguiente capa. Decide a que dispositivo debe enviar el paquete.
- **Drivers de dispositivos de red:** Codifican el tipo de protocolo de diferentes maneras dependiendo de su medio de comunicación.

Los paquetes de IP se entregan al controlador de IP. Este debe decidir hacia donde está dirigido el paquete, y remitirlo al driver de protocolo indicado. El software de IP pasa por *paredes cortafuegos*, filtrando los paquetes para fines de seguridad.
Otras funciones son el desensamblado y reensamblado de paquetes grandes, para poder encolarlos en dispositivos en *fragmentos* más pequeños si no pueden ser encolados por su tamaño.

25.1.11Seguridad

Los problemas de seguridad se pueden clasificar en dos grupos:
- **Validación:** Asegurarse de que solo personas autorizadas puedan ingresar al sistema.
- **Control de acceso:** Proporcionar un mecanismo para verificar si un usuario tiene derechos suficientes para acceder a un objeto dado.

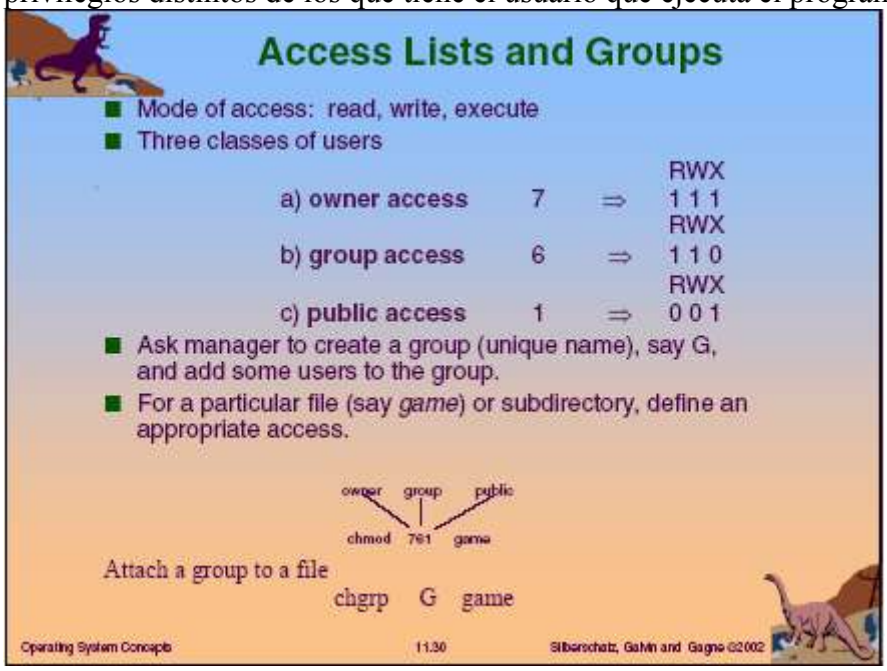
25.1.11.1Validación

La contraseña de un usuario se combina con un valor aleatorio y resultado se codifica con una función de transformación unidireccional (esto implica que no es posible deducir la contraseña original a partir del archivo de contraseña) y se almacena en el archivo de contraseñas. Cuando el usuario la ingresa, se vuelve a codificar, y se busca en el archivo codificado para ver si coincide.

25.1.11.2Control de acceso

Se realiza con la ayuda de identificadores numéricos únicos. El *uid* identifica un solo usuario o un solo conjunto de derechos de acceso, y un *gid* es un identificador de grupo. Cada objeto disponible en el sistema que esta protegido por los mecanismos de control de acceso por usuario y grupo tiene asociado un solo uid y un solo gid.
Linux efectúa el control de acceso a los objetos mediante una *mascara de protección*, que especifica cuales modos de acceso (lectura, escritura, o ejecución) habrán de otorgarse a procesos con acceso de propietario, acceso a grupos o al mundo. Otros de los mecanismos que ofrece es el setuid, el cual permite a un programa ejecutarse con

privilegios distintos de los que tiene el usuario que ejecuta el programa.



26 Windows NT

Es un sistema operativo multitareas, expropiativo de 32 bits para microprocesadores modernos. NT se puede trasladar a varias arquitecturas, y hasta la versión 4 no es multiusuario.

Las dos versiones de NT son Windows NT Workstation y Windows NT Server. Ambas usan el mismo código para el núcleo y el sistema, pero el Server está configurado para aplicaciones cliente-servidor.

26.1.1 Historia

A mediados de los 80's Microsoft e IBM cooperaron para desarrollar OS/2, escrito en lenguaje ensamblador para sistemas monoprocesador Intel 80286.

En 1988 Microsoft decidió desarrollar un sistema portátil que apoyara las interfaces de programación de aplicaciones (API). Se suponía que NT utilizaría la API de OS/2, pero utilizó la Win32 API.

Las primeras versiones fueron Windows NT 3.1 y 3.1 Advanced Server. En la versión 4, NT adoptó la interfaz con el usuario de Windows 95, e incorporó software de servidor de Web en Internet y de navegador. Se pasaron al núcleo algunas rutinas de interfaz con el usuario y código de gráficos a fin de mejorar el desempeño, con el efecto secundario de disminución en la confiabilidad del sistema.

26.1.2 Principios de diseño

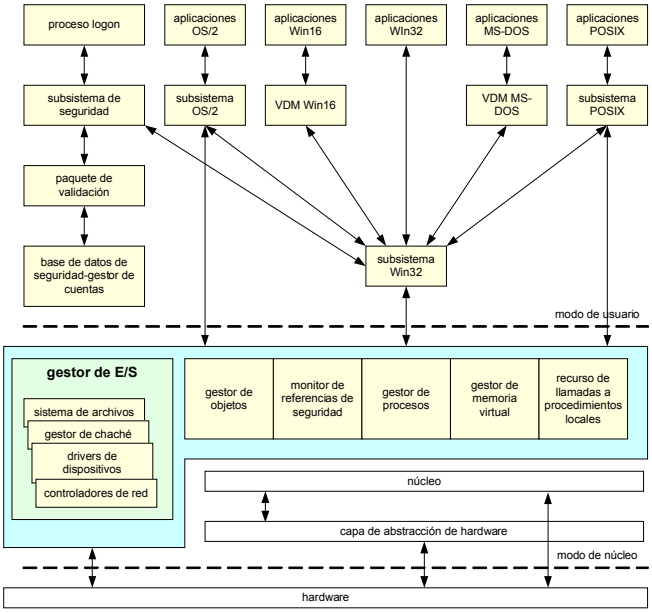
Los objetivos clave del sistema son transportabilidad, seguridad, soporte a multiprocesadores, extensibilidad, soporte internacional y compatibilidad con aplicaciones MS-DOS y MS-Windows.

- **Extensibilidad:** Se implementó con una arquitectura de capas a fin de mantenerse al día con los avances tecnológicos. El ejecutivo NT que opera en modo protegido, proporciona los servicios básicos, y encima operan varios subsistemas servidores en modo usuario, entre los cuales se encuentran los *subsistemas de entorno o ambiente* que emulan diferentes sistemas operativos. Es posible añadir nuevos subsistemas de entorno fácilmente, además NT utiliza controladores cargables, para poder cargar nuevos sistemas de archivos, dispositivos de E/S, etc, mientras el sistema está trabajando.
- **Transportabilidad:** Casi todo el sistema fue escrito en C y C++, al igual que UNIX, y todo el código que depende del procesador está aislado en una DLL llamada *capa de abstracción de hardware (HAL)*. Las capas superiores de NT dependen de la HAL y no del hardware, ayudando a la portabilidad.
- **Confiabilidad:** Emplea protección por hardware para la memoria virtual y por software para los recursos del SO. Posee también un sistema de archivos propio (NTFS) que se recupera automáticamente de muchas clases de errores del sistema de archivos después de una caída del sistema.
- **Compatibilidad:** Ofrece compatibilidad en el nivel de fuente (se compilan sin modificar el código fuente) con aplicaciones que cumplen el estándar IEEE 1003.1 (POSIX). Ejecuta también binarios de muchos programas compilados para Intel X86. Esto se logra mediante los subsistemas de entorno antes mencionados, aunque la compatibilidad no es perfecta, ya que por ejemplo, MS-DOS permitía accesos directos a los puertos de hardware y NT prohíbe tal acceso.
- **Desempeño:** Los subsistemas se comunican de forma eficiente llamando a procedimientos locales que transfieren mensajes con gran rapidez.
- **Uso internacional:** Cuenta con soporte para diferentes ubicaciones a través de la API de soporte a idiomas nacionales (NLS). El código de caracteres nativo es UNICODE, aunque reconoce caracteres ANSI, convirtiéndolos en UNICODE (8 bites a 16 bites).

26.1.3 Componentes del sistema

La arquitectura NT es un sistema de módulos en capas.

Una de las principales ventajas de este tipo de arquitectura es que las interacciones entre los módulos son relativamente sencillas.



26.1.3.1 Capa de abstracción de hardware

Es una capa de software que oculta las diferencias de hardware para que no las perciban los niveles superiores, y lograr portabilidad. Es usada por el núcleo y los drivers de dispositivos, por lo tanto solo se necesita una versión de cada driver, para las diferentes plataformas de hardware.

Por razones de desempeño, los controladores de E/S y de gráficos acceden directamente al hardware.

26.1.3.2 Núcleo

Constituye los cimientos del ejecutivo y los subsistemas. Siempre está cargado en memoria principal.

Tiene cuatro obligaciones principales:

- Planificación de hilos
- Manejo de interrupciones y excepciones
- Sincronización de bajo nivel del procesador
- Recuperación después de una caída en el suministro de electricidad

Está orientado a objetos. Los objetos están definidos por el sistema, y tiene atributos y métodos.

Windows NT, también maneja los conceptos de hilos y procesos. El proceso tiene un espacio de direcciones en memoria virtual, información sobre prioridad, y afinidad por uno o más procesadores.

Cada proceso tiene uno o más hilos, y cada hilo tiene su propio estado, que incluye prioridad, afinidad de procesador e información de contabilidad.

Estados de los hilos:

- **Listo:** Espera para ejecutarse.
- **A punto:** El hilo listo con la prioridad más alta se pasa a este estado, implicando que será el siguiente hilo en ejecutarse. Se mantiene un hilo a punto para cada procesador, en un sistema multiprocesador.
- **Ejecutándose:** Se ejecuta n un procesador hasta ser desalojado por un hilo de más alta prioridad, hasta terminar, hasta que se agote su cuanto de tiempo o hasta que necesite E/S.

- **Esperando:** Espera una señal, como por ejemplo el fin de E/S.
- **Transición:** Espera los recursos necesarios para ejecutarse.
- **Terminado:** Terminó su ejecución.

El despachador tiene un esquema de 32 niveles de prioridad, dividido en dos clases:

- **Tiempo real:** Hilos con prioridad de 16 a 31.
- **Clase variable:** Hilos con prioridad de 0 a 15.

El despachador usa una cola para cada prioridad, y las recorre de mayor a menor, buscando el primer hilo listo a ser ejecutado. Cuando se agota el cuanto de un hilo, se interrumpe, y si el hilo es de prioridad variable, la prioridad se reduce, aunque nunca por debajo de la prioridad base.

Cuando un hilo de prioridad variable sale del estado de espera, se aumenta su prioridad. Esta estrategia tiende a mejorar los tiempos de respuesta de los hilos interactivos que están usando el ratón y las ventanas, manteniendo los dispositivos de E/S ocupados por los hilos limitados por E/S, y que los hilos limitados por CPU, actúen en segundo plano con los ciclos de CPU sobrantes. Además, la ventana actual con la que el usuario está interactuando, recibe un aumento de prioridad para reducir su tiempo de respuesta.

Si un hilo de tiempo real, con prioridad más alta queda listo mientras se ejecuta un hilo con mas baja prioridad, éste es desalojado, por lo tanto la expropiación proporciona acceso preferencial al CPU a los hilos de tiempo real, sin embargo, no se pueden garantizar que se cumplan los límites de tiempo, y por ello NT no es un sistema de tiempo real duro.

El núcleo también maneja trampas para excepciones e interrupciones por hardware o software. NT define excepciones independientes de la arquitectura como violación de acceso a memoria, desbordamiento de enteros, desbordamiento hacia infinito o cero de punto flotante, división entera entre cero, división punto flotante entre cero, instrucción no válida, falta de alineación de datos, instrucción privilegiada, error de lectura de página, violación de página guardia, etc.

El manejador de trampas se encarga de excepciones sencillas, y el *despachador de excepciones* del núcleo se encarga de las demás, buscando manejadores que se encarguen de ellas. Cuando ocurre una excepción en modo de núcleo, si no se encuentra el manejador correcto, ocurre un error fatal del sistema, presentando la “pantalla azul” que significa, fallo del sistema.

El núcleo utiliza una *tabla de despacho de interrupciones* para vincular cada nivel de interrupción con una rutina de servicio, y en un computador multiprocesador, NT mantiene una tabla para cada procesador.

NT se puede ejecutar en máquinas con multiprocesadores simétricos, por lo tanto el núcleo debe evitar que dos hilos modifiquen estructuras de datos compartidas al mismo tiempo. Para lograr la mutua exclusión, el núcleo emplea cerraduras giratorias que residen en memoria global. Debido a que todas las actividades del procesador se suspenden cuando un hilo intenta adquirir una cerradura giratoria, se trata que sea liberada lo más rápido posible.

26.1.3.3 Ejecutivo

Presta un conjunto de servicios que los subsistemas de entorno pueden usar.

26.1.3.3.1 Gestor de objetos

NT es un sistema orientado a objetos, y el gestor supervisa su uso. Cuando un hilo desea usar un objeto, debe invocar el método **open** del gestor de objetos, para obtener un *handle* para ese objeto. Además el gestor se encarga de verificar la seguridad, verificando si el hilo tiene derecho de acceder a ese objeto. El gestor también se mantiene al tanto de que procesos están usando cada objeto. Se encarga de eliminar objetos temporales cuando ya no hay referencias a ellos (Nota: algo del estilo garbageCollector).

26.1.3.3.2 Nombres de objetos

NT permite dar nombres a los objetos. El espacio de nombres es global, por lo tanto un proceso puede crear un objeto con nombre, y otro, puede hacer un handle hacia él, y compartirlo con el primer proceso. El nombre puede ser temporal o permanente. Uno permanente representa entidades, como unidades de disco que existe aunque ningún proceso lo esté usando. Los temporales existen mientras alguien haga referencia a ellos (Nota: no sobreviven al garbageCollector).

Los nombres de objeto tienen la misma estructura que los nombres de camino de archivos en MS-DOS y UNIX. Los directorios se representan con un *objeto directorio*, que contiene los nombres de todos los objetos de ese directorio.

El espacio de nombres de objetos aumenta al agregar un *dominio de objetos*, como por ejemplo discos flexibles y duros, ya que estos tienen su propio espacio de nombres que se debe injertar en el espacio de nombres existente.

Al igual que UNIX, NT implementa un *objeto de enlace simbólico*, refiriéndose a un mismo archivo con varios seudónimos o alias, como por ejemplo la correspondencia de nombres de unidades en NT, con las letras de unidad estándar de MS-DOS.

Cada proceso crea handle a varios objetos, los cuales son almacenados en una *tabla de objetos* del proceso, y al terminar, NT cierra automáticamente todos esos handle.

Cada objeto está protegido por una *lista de control de acceso* que contiene los derechos de acceso. El sistema los compara con los derechos de acceso del usuario que intenta acceder al objeto, y verifica si puede hacerlo. Esta verificación solo se hace al abrir el objeto, y no al crear un handle, por lo tanto los servicios internos de NT que usan handle, pasan por alto la verificación de acceso.

26.1.3.3.3 Gestor de memoria virtual

El diseñador del gestor, supone que el hardware apoya la correspondencia virtual a física, un mecanismo de paginación y coherencia de caché transparente en sistemas multiprocesador, y que permite hacer que varias entradas de la tabla de páginas, correspondan al mismo marco de página.

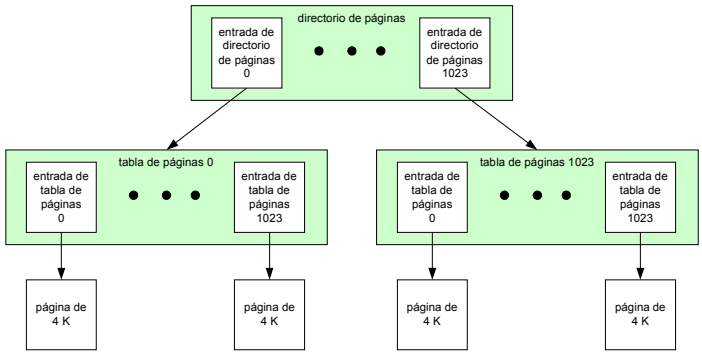
Utiliza un esquema de gestión basado en páginas con un tamaño de 4 KB, y usa direcciones de 32 bits, por lo tanto cada proceso tiene un espacio de direcciones virtual

de 4GB. Los 2 GB superiores son idénticos para todos los procesos, y NT los usa en modo núcleo. Los 2GB inferiores son distintos para cada proceso y son accesibles en modo usuario o núcleo.

Dos procesos pueden compartir memoria obteniendo handle para el mismo objeto, pero sería ineficiente ya que los dos deberían reservar el espacio de memoria, antes de poder acceder al objeto. Para ello se creó el *objeto sección* que representa un bloque de memoria compartida, y que se maneja con handle.

Un proceso puede controlar el uso de un objeto en memoria compartida. Puede protegerse de modo lectura-escritura, solo lectura, solo ejecución, página guardia y copiar al escribir. La página guardia genera una excepción si es accedida, y sirve por ejemplo para controlar si un programa defectuoso va más allá del tope de un arreglo. El mecanismo de copiar al escribir, permite ahorrar memoria, ya que si dos procesos desean copias independientes de un mismo objeto, coloca una sola copia compartida en la memoria, activando la propiedad de copiar al escribir en esa región de memoria, y si uno de los objetos desea modificar los datos, el gestor hace primero una copia privada de la página para que ese proceso la use.

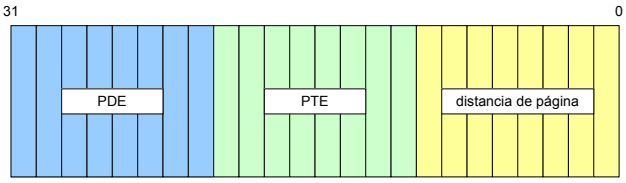
La traducción de direcciones virtuales de NT usa varias estructuras de datos.



Cada proceso tiene un *directorio de páginas* que contiene 1024 *entradas de directorio de páginas (PDE)* de cuatro bytes cada una. El directorio es privado, pero podría compartirse entre procesos si fuera necesario.

Cada entrada de directorio, apunta a una *tabla de páginas* que contiene 1024 *entradas de tabla de páginas (PTE)* de cuatro bytes cada una, y cada *entrada de tabla de páginas* apunta a un *marco de página* de 4 KB en la memoria física.

Una dirección de memoria virtual de 32 bits se divide en tres enteros. Los primeros 10 son el índice de directorio de página, seleccionando la entrada al directorio de páginas que apunta a una tabla de páginas.



Los siguientes 10 bits sirven para escoger una PTE de esa tabla de páginas, la cual apunta a un marco de página en la memoria física.
Los restantes 12 bits se usan para describir la página.

La página puede estar en seis estados:

- **Válida:** Usada por un proceso activo.
- **Libre:** No se hace referencia en ninguna PTE.
- **En ceros:** Página libre, con ceros, lista para usarse de inmediato.
- **A punto:** Ha sido eliminada del conjunto de trabajo de un proceso.

- **Modificada:** Se ha escrito pero todavía no se copió en disco.
- **Defectuosa:** No puede utilizarse porque se detectó un error de hardware.

El principio de *localidad* dice que cuando se usa una página, es probable que en un futuro cercano, se haga referencia a páginas adyacentes, por lo tanto, cuando se produce un fallo de página, el gestor de memoria, también trae las páginas adyacentes, a fin de reducir nuevos fallos.

Si no hay marcos de página disponibles en la lista de marcos libres, NT usa una política de reemplazo FIFO por proceso para expropiar páginas.

NT asigna por defecto 30 páginas a cada proceso. Periódicamente va robando una página válida a cada proceso, y si el proceso se sigue ejecutando sin producir fallos por falta de página, se reduce el conjunto de trabajo del proceso, y se añade la página a la lista de páginas libres.

26.1.3.3.4 Gestor de procesos

Presta servicios para crear, eliminar y usar hilos y procesos. No tiene conocimiento de las relaciones padre-hijo ni de jerarquías de procesos.

26.1.3.3.5 Recurso de llamadas a procedimientos locales

El LPC (local procedure call) sirve para transferir solicitudes y resultados entre procesos clientes y servidores dentro de una sola máquina. En particular se usa para solicitar servicios de los diversos subsistemas de NT. Es un mecanismo de transferencia de mensajes. El proceso servidor publica un objeto *puerto de conexión* que es visible globalmente. Cuando un cliente desea servicios de un subsistema, abre un handle (mango) al objeto puerto de conexión de ese subsistema, y le envía una solicitud de conexión. El servidor crea un canal, y le devuelve un handle al cliente.

El canal consiste en un par de puertos de comunicación privados, uno para los mensajes del cliente al servidor, y otro para los del servidor al cliente.

Hay tres técnicas de transferencia de mensajes para el canal LPC:

- *Mensajes pequeños:* Para mensajes de hasta 256 bytes. Se usa la cola de mensajes del puerto como almacenamiento intermedio, y los mensajes se copian de un proceso al otro.
- *Mensajes medianos:* Se crea un objeto *sección de memoria compartida* para el canal, y los mensajes del puerto, contienen un puntero, e información de tamaño referidos a ese objeto sección, evitando la necesidad de copiar mensajes grandes.
- *Mensajes LPC, o LPC rápida:* Es utilizado por las porciones de exhibición grafica del subsistema Win32. Cuando un cliente pide LPC rápida, el servidor prepara tres objetos: Un hilo servidor, dedicado a atender solicitudes, un objeto sección de 64KB y un objeto *par de sucesos*, que es un objeto de sincronización para Win32, que notifica cuando el hilo cliente copio un mensaje en el servidor o viceversa.
    - *Ventajas:*
        - El objeto sección elimina el copiado de mensajes, ya que representa memoria compartida.
        - El objeto par de sucesos elimina el gasto extra de utilizar el objeto para transferir punteros y longitudes.

- El hilo servidor elimina el gasto extra de determinar que hilo cliente está llamando al servidor ya que hay un hilo servidor por cada cliente.

- *Desventajas:*
  - Se utilizan más recursos que con los otros métodos, por lo tanto solo se usa para el gestor de ventanas e interfaces con dispositivos de gráficos.

26.1.3.3.6 Gestor de E/S:

Funciones:

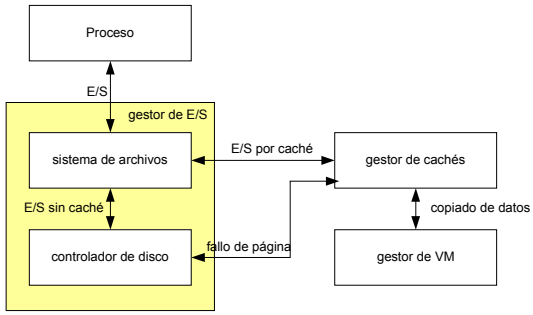
- Se mantiene al tanto de cuales sistemas de archivos instalables están cargados.
- Gestiona buffers para solicitudes de E/S
- Trabaja con el gestor de cachés de NT que maneja los cachés de todo el sistema de E/S.
- Trabaja con el gestor de Memoria Virtual para efectuar E/S de archivos con correspondencia en la memoria.
- Apoya operaciones de E/S, sincrónicas y asincrónicas.
- Proporciona tiempos límite para los controladores, y cuenta con mecanismos para que un controlador llame a otro.

En muchos sistemas operativos, el sistema de archivos se encarga de manejar los caché, en cambio NT cuenta con un manejo de cachés centralizado. El gestor de caché presta servicios, controlado por el gestor de E/S, e íntimamente vinculado con el gestor de VM.

Los 2GB superiores del espacio de direcciones de un proceso, son del sistema, y el gestor de VM asigna la mitad del espacio disponible al caché del sistema.

Cuando el gestor de E/S recibe una solicitud de lectura, envía un IRP al gestor de cachés (a menos que se solicite lectura sin uso de caché). El gestor de caché calcula la dirección de la solicitud, y pueden suceder dos cosas con el contenido de la dirección:

- **Es nula:** asigna un bloque al caché e intenta copiar los datos invocados.
  - Se puede realizar el copiado: La operación finaliza.
  - No se puede realizar el copiado: Se produjo un fallo de página, lo que hace que el gestor de VM envíe una solicitud de lectura sin uso del caché. El driver apropiado, lee los datos, y los devuelve al gestor de VM, que carga los datos en el caché, y los devuelve al invocador, quedando satisfecha la solicitud.



- **No es nula:** Se leen los datos y la operación finaliza.

Para mejorar el desempeño, el gestor mantiene un historial de solicitudes de lectura, e intenta predecir solicitudes futuras. Si detecta un patrón en las tres solicitudes anteriores, como acceso secuencial hacia delante o atrás, obtiene datos y los coloca en el caché, antes de que la aplicación haga la siguiente solicitud, evitando la espera de leer a disco.

El gestor también se encarga de indicar al gestor de VM que escriba de vuelta en disco el contenido del caché. Efectúa escritura retardada, ya que acumula escrituras durante 4 o 5 segundos, y luego despierta el hilo escritor. Si sucede que un proceso escribe rápidamente llenando las páginas libres del caché, el gestor está controlando, y cuando se llena gran parte de las páginas, bloquea los procesos que escriben datos, y despierta el hilo escritor.

En el caso de redes, para que no queden procesos bloqueados en la red (causando retransmisión por exceso de tiempo), se puede ordenar al gestor de caché que no deje que se acumulen muchas escrituras en el caché. Hay una interfaz DMA directa entre el caché y la red, evitando el copiado de datos a través de un buffer intermedio.

26.1.3.3.7 Gestor de referencias de seguridad

La naturaleza orientada a objetos de NT permite la validación de accesos durante la ejecución y verificaciones de auditoria para cada entidad del sistema.

26.1.4 Subsistemas de entorno

Son procesos en modo usuario, apilados sobre los servicios nativos del ejecutivo de NT, que permiten ejecutar programas escritos para otros sistemas operativos, incluidos Windows de 16 bits, MS-DOS, POSIX y aplicaciones basadas en caracteres para el OS/2 de 16 bits. Cada subsistema de entorno, proporciona una API.

NT usa el subsistema Win32 como entorno operativo principal. Si el ejecutable no es Win32 nativo, se verifica si ya se está ejecutando el subsistema de entorno apropiado, y si no se está ejecutando, se pone en marcha como proceso en modo usuario y Win32 le transfiere el control a dicho subsistema.

El subsistema de entorno no accede directamente al núcleo, contribuyendo a la robustez de NT, ya que se permite controlar la correctitud de los parámetros antes de invocar la rutina del núcleo. NT prohíbe a las aplicaciones, mezclar rutinas de API de diferentes entornos.

La caída de un subsistema, no afecta a los demás, salvo el caso de Win32 que es el gestor de todos los demás.

Win32 clasifica las aplicaciones como gráficas o de caracteres (se supone que las salidas se dirigen a una pantalla ASCII de 80x24). Cuando es de caracteres, transforma la salida, en una representación gráfica dentro de una ventana (Caso de la ventanita de MS-DOS, en el escritorio de Windows). Al ser una emulación, se puede transferir texto de pantalla entre ventanas, a través del portapapeles.

26.1.4.1 Entorno MS-DOS

MS-DOS no tiene la complejidad de los otros subsistemas de entorno NT. Una aplicación Win32 llamada *máquina dos virtual (VDM)* se encarga de proporcionar el entorno. La VDM tiene una *unidad de ejecución de instrucciones* para ejecutar o emular instrucciones Intel 486, también el ROM BIOS de MS-DOS y los servicios de interrupción de software “int 21”, así como drivers de dispositivos virtuales para pantalla, teclado y puertos de comunicación.

Está basada en el código fuente de MS-DOS 5.0 y proporciona a la aplicación al menos 620 KB de memoria. El shell es una ventana muy similar a un entorno MS-DOS.

Si NT se ejecuta en un procesador X86, las aplicaciones gráficas de MS-DOS se ejecutan en modo de pantalla completa, y las de caracteres en cualquiera de las dos opciones (completa o ventana). Si la arquitectura de procesador es otra, todas las aplicaciones se ejecutan en ventanas. En general, las aplicaciones MS-DOS que acceden directamente a hardware, no funcionan bajo NT.

Hay aplicaciones para MS-DOS que acaparan la CPU (ya que no era multitareas) como espera activa o pausas. El despachador de NT detecta los retardos, y los frena (haciendo que la aplicación no funcione correctamente).

26.1.4.2 Entorno Windows de 16 bits

Corre por cuenta de un VDM que incorpora software adicional llamado *windows on windows* que cuenta con las rutinas del núcleo de Windows 3.1, y las rutinas adaptadoras (stubs) para las funciones del gestor de ventanas. Las rutinas adaptadoras, invocan las subrutinas apropiadas de Win32. Las aplicaciones que dependen de la estructura interna del gestor de ventanas de 16 bits no funcionan, porque Windows on Windows no implementa realmente la API de 16 bits.

Windows on Windows puede ejecutarse en forma multitareas con otros procesos en NT, pero solo una aplicación Win16 puede ejecutarse a la vez. Todas las aplicaciones son de un solo hilo, y residen en el mismo espacio de direcciones, con la misma cola de entrada.

26.1.4.3 Entorno Win32

Ejecuta aplicaciones Win32 y gestiona toda la E/S de teclado, ratón y pantalla. Es robusto, y a diferencia de Win16, cada proceso Win32 tiene su propia cola de entrada.

El núcleo NT ofrece operación multitareas expropiativa, que permite al usuario terminar aplicaciones que han fallado o no se necesitan.

Win32 valida todos los objetos antes de usarlos, evitando caídas, el gestor de objeto, evita que se eliminen objetos que todavía se están usando, e impide su uso, una vez eliminados.

26.1.4.4 Subsistema POSIX

Está diseñado para ejecutar aplicaciones POSIX, basadas en el modelo UNIX. El estándar no especifica impresión, pero se pueden usar impresoras a través del mecanismo de redirección de NT.

Las aplicaciones POSIX tienen acceso a cualquier sistema de archivos NT, pero hace respetar los permisos tipo UNIX en los árboles de directorio. No apoya varios recursos de Win32 como trabajo en redes, gráficos, etc.

26.1.4.5 Subsistema OS/2

NT proporciona recursos limitados en el subsistema de entorno OS/2, aunque en sus el propósito general había sido proporcionar un entorno OS/2 robusto.

Las aplicaciones basadas en caracteres solo pueden ejecutarse bajo NT en computadores Intel x86. Las de modo real, se pueden ejecutar en todas las plataformas, usando el entorno MS-DOS.

26.1.4.6 Subsistemas de ingreso y seguridad

Antes de que un usuario pueda acceder a objetos NT, el subsistema de ingreso debe validarlo (login). Para ello el usuario debe tener una cuenta, y proporcionar la contraseña.

Cada vez que el usuario intenta acceder a un objeto, el subsistema chequea que se tengan los permisos correspondientes.

26.1.5 Sistema de archivos

Los sistemas MS-DOS utilizaban el sistema de archivos FAT, de 16 bits, que tiene muchas deficiencias, como fragmentación interna, una limitación de tamaño de 2 GB, y falta de protección de archivos. FAT32 resolvió los problemas de tamaño y fragmentación, pero su desempeño y funciones son deficientes en comparación con NTFS.

NTFS se diseñó teniendo en mente muchas funciones, incluidas recuperación de datos, seguridad, tolerancia de fallos, archivos y sistemas de archivos grandes, nombres UNICODE y compresión de archivos. Por compatibilidad también reconoce los sistemas FAT.

26.1.5.1 Organización interna

La entidad fundamental de NTFS es el volumen. El administrador de disco de NT crea los volúmenes que se basan en particiones de disco lógicas. Un volumen puede ocupar una porción de disco, un disco entero o varios discos.

No maneja sectores de disco individuales sino que usa *cúmulos* (número de sectores de disco que es potencia de 2) como unidad de asignación. El tamaño del cúmulo se configura cuando se formatea el sistema de archivos NTFS.

Tamaños por omisión:

- Tamaño de sector en el caso de volúmenes de hasta 512 Mb.
- 1 KB para volúmenes de hasta 1 GB
- 2 KB para volúmenes de hasta 2 GB
- 4 KB para volúmenes mayores

Este tamaño es mucho menor al del cúmulo de FAT, ayudando a reducir la fragmentación interna. Ej: En un disco de 1.6 GB con 16000 archivos, se podrían perder 400MB con sistema FAT (cúmulos de 32KB), o 17MB bajo NTFS.

NTFS usa *números de cúmulo lógicos (LCN),* y los asigna numerando los cúmulos desde principio de disco, al final, pudiendo calcular la distancia física en el disco, multiplicando el LCN por el tamaño del cúmulo.

Un archivo NTFS, no es un simple flujo de bytes como en MS-DOS o UNIX, sino un objeto estructurado con atributos, algunos estándar, y otros específicos para ciertas clases de archivos.

El espacio de nombres está organizado como una jerarquía de directorios, y cada directorio emplea una estructura llamada *arbol B+*, debido a que elimina el costo de reorganizar el árbol, y tiene la propiedad de que la longitud de cada camino desde la raíz hasta una hoja, es la misma. La *raíz índice* de un directorio, contiene el nivel superior del árbol. Si el directorio es grande, el nivel superior contiene punteros a alcances de disco que contienen el resto del árbol.

Cada entrada de directorio contiene el nombre y la referencia del archivo y una copia del tiempo de actualización, y del tamaño. Esto es útil para no tener que recabar en la MFT (*Tabla maestra de archivos*) cuando se pide un listado de los archivos del directorio.

Los datos del volumen NTFS se almacenan en archivos. El primero es la MFT, y el segundo contiene una copia de las 16 primeras entradas de la MFT, que se utiliza en la recuperación si la MFT se daña.

Otros archivos:

- **Archivo de bitácora:** Registra actualizaciones de los datos del sistema de archivos.
- **Archivo de volumen:** Contiene el nombre del volumen, la versión de NTFS que dio formato, y un bit que indica si es posible que el volumen se haya corrompido, y se necesite verificar su consistencia.
- **Tabla de definición de atributos:** Indica los atributos del volumen y que operaciones se pueden efectuar con cada uno de ellos.
- **Directorio raíz:** Es el directorio del nivel más alto en la jerarquía del sistema de archivos.
- **Archivo de mapa de bits:** Indica que cúmulos del volumen están asignados, y cuales están libres.
- **Archivo de arranque:** Contiene el código de inicio de NT, y debe estar en una dirección específica para que el autoarranque de la ROM lo localice fácilmente. También contiene la dirección de la MFT.
- **Archivo de cúmulos defectuosos:** Sigue la pista a las áreas defectuosas del volumen, y se usa para recuperación de errores.

26.1.5.2 Recuperación

En NTFS, todas las actualizaciones de estructura de datos, se efectúan en *transacciones.* Antes de alterar la estructura, se escribe un registro de bitácora con información para deshacerla y rehacerla, y luego de modificarla con éxito, se escribe un registro de confirmación en la bitácora.

Después de una caída el sistema puede restaurar las estructuras, procesando la bitácora, rehaciendo las operaciones de las transacciones confirmadas, y deshaciendo la de las que no lograron confirmarse antes de la caída.

Periódicamente (cada cinco segundos aproximadamente), se escribe en la bitácora un checkpoint, indicando al sistema que no necesita registros de bitácora anteriores a él, desechando los registros, para que la bitácora no crezca indefinidamente.

La primera vez que se accede a NTFS luego de una caída, se realiza automáticamente la recuperación del sistema de archivos. Esto no garantiza la correctitud de los archivos de usuario, solo asegura las estructuras de datos del sistema de archivos.

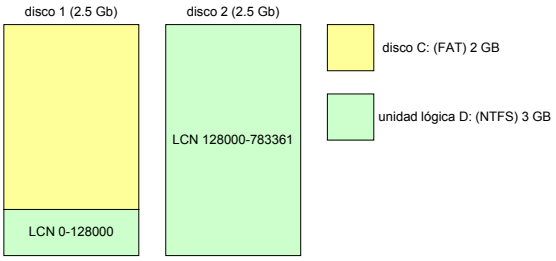
Si el espacio de la bitácora se reduce demasiado, NTFS suspende todas las E/S nuevas, para poder confirmar las que se están haciendo, poner un checkpoint en la bitácora, y borrar todos los datos ya confirmados.

26.1.5.3 Seguridad

La seguridad se deriva del modelo de objetos, ya que cada uno tiene almacenado un descriptor de seguridad.

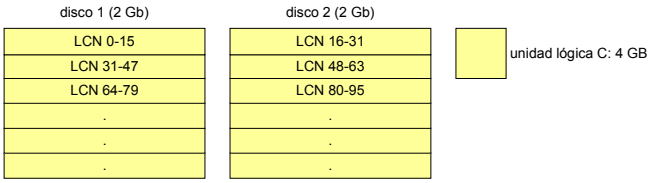
26.1.5.4 Gestión de volúmenes y tolerancia de fallos

FtDisk es el controlador de disco tolerante a fallos. Ofrece mecanismos para combinar múltiples unidades de disco en un solo volumen lógico. Hay varias formas de hacerlo.



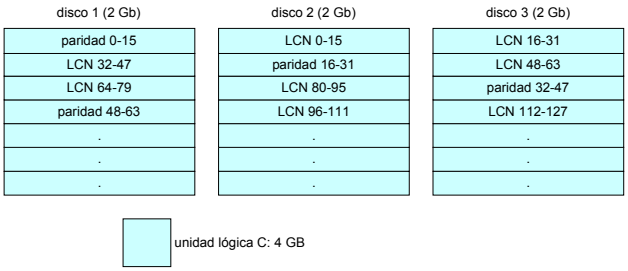
Se pueden concatenar discos lógicamente para formar un volumen lógico grande. En NT es llamado *conjunto de volúmenes* y puede consistir de hasta 32 unidades físicas. NTFS usa el mismo mecanismo LCN que para un solo disco físico, y FtDisk se encarga de hacer la correspondencia.

Otra forma de combinar múltiples particiones físicas es intercalar sus bloques



bajo un régimen de turno circular, para formar un *conjunto de franjas*. FtDisk usa 64Kb como tamaño de franja. Los primeros 64Kb de volumen lógico se almacenan en la primer partición física, los segundos en la segunda y así hasta que cada partición física tiene 64Kb de la lógica.

Esto permite mejorar el ancho de banda de E/S ya que las unidades pueden transferir datos en paralelo.



Se pueden dividir también en *franjas de paridad*. Si hay 8 discos, entonces por cada siete franjas de datos en siete discos diferentes hay una franja de paridad en el octavo que contiene el **or exclusivo** de las siete franjas, por lo tanto, si una se destruye, se puede reconstruir. De todas formas, cada actualización de una franja de datos, obliga a recalcular la franja de paridad, así siete escrituras concurrentes, requieren calcular siete franjas de paridad, por lo tanto, se deben encontrar en discos diferentes, para no sobrecargar un mismo disco con el cálculo de todas las paridades. Las paridades se asignan entre los discos por turno circular.

Un esquema más robusto, es el *espejo de discos*. Un *juego de espejos* comprende dos particiones del mismo tamaño, en dos discos, con su contenido de datos idéntico. Cuando se requiere escritura, FtDisk escribe en los dos discos, y si hay fallo, se tiene una copia almacenada a salvo. Los dos discos, deben estar conectados a dos controladores de disco diferentes, por si el fallo es en el controlador. Esta organización se denomina *juego duplex*.

Para superar la situación en que sectores del disco se arruinan, FtDisk usa una técnica de hardware llamada *ahorro de sectores*, y NTFS, una de software llamada *retransformación de cúmulos.*

Cuando se da formato al disco, se crea correspondencia entre números de bloque lógicos y sectores libres de defectos del disco, dejando algunos sectores sin correspondencia como reserva. Si un sector falla, FtDisk ordena a la unidad de disco que lo sustituya por una unidad de reserva.

La retransformación de cúmulos se aplica por NTFS, que reemplaza un bloque defectuoso por uno no asignado, modificando los punteros pertinentes en la MFT, además tiene en cuenta que ese bloque no vuelva a ser asignado.

Si un bloque se arruina, se pierden datos, que en casos se pueden calcular si se tiene un disco espejo, o calculando la paridad por or exclusivo, y los datos reconstruidos se colocan en un nuevo espacio, que se obtuvo por ahorro de sectores o retransformación de cúmulos.

26.1.5.5 Compresión

NTFS puede aplicar compresión a archivos. Divide sus datos en *unidades de compresión*, que son bloques de 16 cúmulos contiguos. Cada vez que se escribe en una unidad de compresión, se aplica un algoritmo de compresión de datos. Para mejorar la lectura, NTFS las preobtiene y descomprime adelantándose a las solicitudes de las aplicaciones.

En caso de archivos *poco poblados* (contiene gran cantidad de ceros), la técnica es no almacenar en disco los cúmulos que contienen únicamente ceros. Se dejan huecos en la secuencia de números almacenados en la tabla MFT para ese archivo, y al leerlos, si NTFS encuentra un hueco en los números, llena con ceros esa porción del buffer del invocador. UNIX emplea la misma técnica.

26.1.6 Trabajo con redes

NT cuenta con recursos de gestión de redes, y apoya el trabajo de cliente-servidor. Además también tiene componentes de trabajo con redes como:

- Transporte de datos
- Comunicación entre procesos
- Compartimiento de archivos a través de una red
- Capacidad para enviar trabajos de impresión a impresoras remotas.

26.1.6.1 Protocolos

Implementa los protocolos de transporte como controladores, los cuales puede cargar y descargar dinámicamente, aunque en la práctica, casi siempre debe reiniciarse el sistema después de un cambio.

26.1.6.2 Mecanismos de procesamiento distribuido

NT no es distribuido, pero si apoya aplicaciones distribuidas. Mecanismos apoyados:

- **Aplicaciones NetBIOS:** Se comunican en NT a través de la red utilizando algún protocolo.
- **Conductos con nombre:** Son mecanismos de envío de mensajes, orientados a conexiones. Un proceso también puede usarlos para conectarse con otro proceso en la misma máquina. Puesto que el acceso a estos conductos es a través de la interfaz con el sistema de archivos, se aplican los mecanismos de seguridad sobre ellos.
- **Ranuras de correo:** Son un mecanismo de envío de mensajes sin conexiones. No son confiables, porque el mensaje podría perderse antes de llegar al destinatario.
- **Sockets:** Winsock es la API de sockets para ventanas. Es una interfaz de capa de sesión, compatible (en buena parte) con sockets de UNIX.
- **Llamada a procedimiento remoto (RPC***):* Es un mecanismo cliente-servidor que permite a una aplicación de una máquina, emitir una llamada a un procedimiento cuyo código está en otra máquina. El cliente invoca, un stub empaca sus argumentos en un mensaje y lo envía por la red a un proceso servidor específico. El servidor desempaca el mensaje, invoca el procedimiento, empaca los resultados en otro mensaje, y lo envía de vuelva al cliente, que espera el mensaje bloqueado. El cliente se desbloquea, lo recibe, y desempaca los resultados.
- **Lenguaje de definición de interfaces de Microsoft:** Se utiliza para describir los nombres de procedimientos remotos, argumentos y resultados, para facilitar

las llamadas a procedimientos remotos. El compilador para este lenguaje genera archivos de cabecera que declaran rutinas adaptadoras para los procedimientos remotos, asi como un desempacador y un despachador.

- **Intercambio dinámico de datos (DDE):** Mecanismo para la comunicación entre procesos creado para Microsoft Windows.

26.1.6.3 Redirectores y servidores

En NT, una aplicación puede usar la API de E/S para acceder a archivos remotos. Un *redirector* es un objeto del lado del cliente que reenvía solicitudes de E/S a archivos remotos, donde un *servidor* las atiende. Por razones de desempeño y seguridad, los redirectores y servidores se ejecutan en modo de núcleo. Por razones de transportabilidad, redirectores y servidores utilizan la API TDI para el transporte por redes.

26.1.6.4 Dominios

Para gestionar derechos de acceso globales (ej: todos los usuarios de un grupo de estudiantes), NT utiliza el concepto de dominio.

Un dominio NT, es un grupo de estaciones de trabajo y servidores NT que comparten una política de seguridad y una base de datos de usuarios comunes. Un servidor de dominio controla y mantiene las bases de datos de seguridad. Otros servidores pueden actuar como controladores de respaldo, y validación de solicitudes de inicio.

A menudo es necesario manejar múltiples dominios dentro de una sola organización. NT ofrece cuatro modelos de dominios para hacerlo.

- **Modelo de dominio único:** Cada dominio queda aislado de los demás.
- **Modelo de dominio maestro:** Todos los dominios confían en un dominio maestro designado, que mantiene las bases de datos de cuentas de usuario, y proporciona servicios de validación. Puede apoyar hasta 40000 usuarios.
- **Modelo de múltiples dominios maestros:** Diseñado para organizaciones grandes, con más de un dominio maestro, y todos los dominios confían unos en otros. El usuario con cuenta en un dominio, puede acceder a recursos de otro dominio, si ese dominio certifica los permisos del usuario.
- **Modelo de múltiple confianza:** No existe un dominio maestro, todos los dominios confían unos en otros.

26.1.6.5 Resolución de nombres en redes TCP/IP

Es el proceso de convertir un nombre de computador en una dirección de IP. Por ejemplo, transformar www.bell-labs.com en 135.104.1.14.

NT cuenta con varios métodos, incluidos el Servicio de Nombres de Internet de Windows (WINS), resolución de nombres difundida, sistema de nombres de dominio (DNS).

WINS: Dos o más servidores WINS, mantienen una base de datos dinámica de vinculaciones Nombre-dirección de IP, y software cliente para consultarlos. Se usan dos servidores para que el servicio funcione, aunque falle uno, y además para distribuir la carga.

26.1.7 23.7 Interfaz con el programador

La API Win32 es la interfaz fundamental con las capacidades de NT.

26.1.7.1 23.7.1 Acceso a objetos del núcleo

Se acceden para obtener los servicios del núcleo. Cuando un proceso quiere acceder a un objeto X del núcleo, debe hacer **createX** y abre un handle a ese objeto. Para cerrar el handle hace **closeHandle** y el sistema elimina el objeto si nadie tiene handles con él.

26.1.7.1.1 Compartimiento de objetos

Hay tres formas:

- Un proceso hijo hereda un handle del padre. El padre y el hijo podrán comunicarse a través del objeto.
- Un proceso crea un objeto con nombre, y el segundo proceso lo abre con ese nombre.
    - Desventajas:
        - NT no ofrece un mecanismo para verificar si existe un objeto con nombre escogido.
        - El espacio de nombres es global. Por ejemplo, dos aplicaciones podrían crear un objeto llamado pepe, cuando se desean dos objetos individuales (y diferentes).
    - Ventajas:
        - Procesos no relacionados entre si, pueden compartir objetos fácilmente.
- Compartir objetos por medio de la función **DuplicateHandle**. Pero requiere de un método de comunicación entre procesos para pasar el mango duplicado.

26.1.7.2 Gestión de procesos

Un proceso es un ejemplar de una aplicación que se está ejecutando y un hilo es una unidad de código que el sistema operativo puede planificar. Un proceso tiene uno o más hilos.

Una aplicación multihilada necesita protegerse contra acceso no sincronizado, y la función envolvente **beginthreadex** (iniciar ejecución de hilo) proporciona la sincronización apropiada.

Las prioridades en el entorno Win32 se basan en el modelo de planificación NT, pero no pueden escogerse todos los valores de prioridad. Se usan cuatro clases:

- **IDLE_PRIORITY_CLASS:** Nivel de prioridad 4 (ocioso).
- **NORMAL_PRIORITY_CLASS:** Nivel de prioridad 8 (normal).
- **HIGH_PRIORITY_CLASS:** Nivel de prioridad 13 (alta).
- **REALTIME_PRIORITY_CLASS:** Nivel de prioridad 24 (tiempo real).

Solo los usuarios con *privilegio de incremento de prioridad* pueden pasar un proceso a la prioridad de tiempo real. Por defecto los administradores o usuarios avanzados tienen ese privilegio.

Cuando un usuario está ejecutando un programa interactivo, el sistema debe darle mayor prioridad, para ello tiene una regla de planificación que distingue entre el *proceso de primer plano* que es el seleccionado actualmente en la pantalla y los *procesos de segundo plano* que no están seleccionados. Cuando un proceso pasa a primer plano, el sistema incrementa su cuanto de planificación al triple, por lo que el proceso se ejecuta por un tiempo tres veces mayor, antes de ser desalojado.

Para sincronizar el acceso concurrente de hilos a objetos compartidos, el núcleo proporciona objetos de sincronización como semáforos y mutexes. Otro método es la *sección crítica*, que es una región de código sincronizada que solo un hilo puede ejecutar a la vez.

Una *fibra* es código en modo usuario, planificado según un algoritmo definido por el usuario. Se utiliza de la misma forma que los hilos, pero a diferencia de ellos, no se pueden ejecutar dos fibras concurrentemente, aún en un hardware multiprocesador.

26.1.7.3 Comunicación entre procesos

Una forma es compartiendo objetos del núcleo, y otra es transfiriendo mensajes.

Los mensajes se pueden publicar o enviar. Las rutinas de publicación son asincrónicas, por lo tanto envían el mensaje y regresan de inmediato. El hilo que envió el mensaje no sabe si llegó. Las rutinas de envío son sincrónicas, y bloquean al invocador hasta que se ha entregado y procesado el mensaje.

Cada hilo Win32 tiene su propia cola de entrada, y esta estructura es más confiable que la cola de entradas compartida de las ventanas de 16 bits, porque una aplicación bloqueada no bloquea a las demás.

Si un proceso que envió un mensaje, no recibe respuesta en cinco segundos, el sistema marca la aplicación como “No responde”.

26.1.7.4 Gestión de memoria

Hay varios mecanismos de uso de memoria como:

- **Memoria Virtual:** Para pedir y liberar memoria virtual se usa **VirtualAlloc** y **VirtualFree**. La función **VirtualLock** asegura páginas en memoria física, pudiendo asegurar 30 como máximas.

- **Archivos con correspondencia en memoria:** Es cómodo para que dos procesos compartan memoria.
- **Montículo (heap):** Es una región de espacio de direcciones reservada. Cuando Win32 crea un proceso, lo hace con un *montículo por omisión* de 1MB. El acceso al montículo es sincronizado para proteger las estructuras de datos de hilos concurrentes. Se usan funciones como HeapCreate, HeapAlloc, HeapRealloc, HeapSize, HeapFree y HeapDestroy.

27 Dudas:

-hay MMU cuando no tenemos memoria virtual?

-puedo consultar el estado de una condicion en un monitor? (status)

-el directorio al que se refieren en file system (que tiene almacenadas las propiedades de los archivos) es el directorio en donde se guardan los archivos como /home/usuarioX o es una estructura de datos distinta?

Preguntas interesantes:

En el estudio de la segmentación simple, se llegó a la conclusión de que cada proceso tiene su propia tabla de segmentos

• *Tabla de páginas:* Normalmente, hay una tabla por proceso, con una entrada para cada página de la memoria virtual del proceso.

• *Descriptor de bloques de disco:* Asociado a cada página de un proceso hay una entrada en la tabla que describe la copia en disco de la página virtual.

Un *procedimiento reentrante* es aquel en el que sólo una copia del código del programa puede estar compartida entre varios usuarios durante el mismo periodo.

cercanía de referencias [DENN68]. Durante el curso de ejecución de un programa, las referencias a me-moria por parte del procesador, tanto para instrucciones como para datos, tienden a estar agrupadas. Los programas contienen normalmente un cierto número de bucles y de subruti-
nas iterativas. Una vez que se entra en un bucle o en una subrutina, se producirán referencias repetidas a un pequeño conjunto de instrucciones. De forma similar, las operaciones sobre tablas y vectores suponen el acceso a un conjunto de palabras de datos que están agrupadas. Durante un largo período, las agrupaciones en uso cambian, pero, en un período corto, el procesador trabaja principalmente con grupos fijos de referencias a memoria.

En un sistema con memoria virtual el espacio logico de un proceso puede ser mayor que la memoria real disponible pero no tiene porque serlo necesariamente.

Poner lo de la pagina 298 **del libro** sta